

Джордж Шеферд
по материалам Дэвида Круглински



Программирование Microsoft® на VISUAL C++ .NET

Издание второе

Microsoft® .net

РУССКАЯ РЕДАКЦИЯ Microsoft® ПИТЕР®

Оглавление

ЧАСТЬ II ОСНОВЫ MFC 27

Глава 16 Чтение и запись документов 339

ЧАСТЬ 4
COM, AUTOMATION, ACTIVEX И OLE 459

Глава 22 Модель компонентных объектов 460

Глава 23 Automation 497

Глава 34 Программирование в ASP.NET на управляемом C++ 827

Глава 35 Программирование в ADO.NET на управляемом C++ 851

Приложение А Функции карт сообщений в библиотеке MFC 865

Приложение Б Идентификация MFC-классов во время выполнения и динамическое создание объектов 871

Предметный указатель 877

Об авторе 893

Благодарности

Писать этот раздел приятнее всего — работа над книгой практически завершена, и остается лишь поблагодарить всех, кто приложил свои силы, чтобы она стала реальностью. Имя автора выведено большими буквами на обложке, и поэтому о том огромном числе людей, которые приняли участие в создании книги, отдав ей массу своего времени и энергии, часто забывают. Так что я хочу поблагодарить этих людей.

Я хочу сказать спасибо Сэнди Дастон (Sandy Daston) и Теду Шеферду (Ted Shepherd) — это моя семья — за поддержку во время написания книги.

Спасибо Дениз Банкаитис (Denise Bankaitis), которая выполняла в проекте функции редактора и постоянно напоминала мне о важности книги (ведь это один из ключевых справочников по программированию на C++ для .NET), а также координировала деятельность команды, всем членам которой — отдельное спасибо:

Джули Сяо (Julie Xiao) — за заботу по искоренению ошибок;

Айне Ченг (Ina Chang) — за то, что написанные мной предложения стали читабельными;

Дэниэлу Берду (Danielle Bird) и Джулиане Олдос (Juliana Aldous) — рецензентам издательства, поддерживавшим проект и не позволившим ему сбиться с пути;

Джоэлу Пенчоту (Joel Panchot) — за то, что графика в книге прекрасно смотрится;

Карлу Дицу (Carl Diltz) и Джине Кассилл (Gina Cassill) — за создание макета книги и приведение ее в удобоваримый вид.

Хотелось бы также поблагодарить сотрудников учебного центра DevelopMentor за организацию сообщества единомышленников и создание прекрасных условий обсуждения и изучения современных информационных технологий. Вы — самые лучшие.

Введение

Выход Microsoft Visual Studio .NET (в частности, Visual C++ .NET) подтвердил взятый компанией Microsoft курс на Интернет-технологии, которые легли в основу архитектуры Microsoft .NET. Кроме поддержки инициативы .NET, в Visual C++ .NET есть полный набор средств повышения производительности труда программиста, в том числе уже знакомые вам Edit And Continue, IntelliSense, AutoComplete и подсказки по коду (code tips). В Visual C++ .NET также масса новых возможностей, таких как управляемый C++ для программирования приложений .NET, поддержка кода на основе атрибутов и более продуманная и теснее интегрированная среда разработки, которые ставят Visual C++ .NET на более высокий уровень. Эта книга поможет вам не отстать от технологий, появившихся в Visual C++ .NET.

.NET, MFC и ATL

Технологии в современном мире развиваются непостижимо быстро. Сначала ни у кого не было настольных компьютеров, в 80-х почти все обзавелись машинами под управлением MS-DOS, и, наконец, к середине 90-х компьютеры под управлением Microsoft Windows заполонили мир. И, по-видимому, технологии сделают еще один виток. В конце 90-х Web-сайты разрабатывались вручную, с использованием «простого» языка HTML (Hypertext Markup Language), CGI (Common Gateway Interface), DLL-библиотек ISAPI (Internet Server Application Programming Interface), Java и ASP-страниц (Active Server Pages). В июле 2000 г. Microsoft возвестила о намерении заменить все это технологией .NET.

Сейчас Microsoft вовсю работает над .NET. Пару лет назад для создания Web-сайта нужно было лишь установить сервер, приобрести IP-адрес и «выложить» на сайт какую-то информацию. После этого сайт становился доступен всем — достаточно было знать URL-адрес. Коммерческие предприятия использовали Web для размещения данных, которые могли пригодиться клиентам. Web-среда также оказалась ценным исследовательским инструментом и средством распространения информации.

В ИТ-мире ближайшего будущего Web будет играть первую скрипку. Однако ситуация изменится: до этого содержимое Web-сайтов предназначалось пользователям, теперь же с этой информацией будут работать другие компьютеры. То есть доступ к содержимому Web-сайтов станет возможен из программ — благодаря Web-сервисам. Согласно концепциям .NET ответственность за организацию многофункционального пользовательского интерфейса возлагается на сервер.

В условиях эйфории относительно Web-сервисов и пользовательских интерфейсов, поддерживаемых сервером, может показаться, что автономные приложения и клиентские сценарии — прерогатива таких средств, как библиотека MFC

(Microsoft Foundation Class Library), — окажутся за бортом истории. Однако вряд ли исчезнет потребность в полнофункциональном пользовательском интерфейсе. Многие полагали, что «персоналки» и распределенные технологии естественным путем вытеснят мейнфреймы и миникомпьютеры, а оказалось, что ПК и распределенные вычисления лишь дополнили общую картину ИТ-мира. Принципы Web-сервисов в .NET и поддерживаемые сервером многофункциональные пользовательские интерфейсы стали еще одним вариантом, доступным разработчикам. Полнофункциональные пользовательские интерфейсы никуда не уйдут из большинства приложений, которые прекрасно уживутся с другими приложениями с другого типа интерфейсами (в том числе поддерживаемыми сервером Web-интерфейсы).

MFC — зрелый, хорошо изученный инструмент, обеспеченный обширной поддержкой сторонних компаний. Какое-то время эта библиотека останется одним из самых производительных способов создания полнофункциональных автономных приложений. Львиная доля данной книги посвящена приложениям «в стиле MFC», но мы расскажем и о Windows Forms — технологии создания интерфейсов клиентских приложений в .NET.

А что будет с COM? Эта технология позволила решить массу проблем организации распределенных вычислений, но у нее есть ряд серьезных недостатков, как правило, связанных с поддержкой версий и информации о типах. Работа .NET основана на *общеязыковой среде исполнения*, или CLR (Common Language Runtime). Она берет на себя функции COM по обеспечению совместимости. О .NET и CLR-среде подробно рассказывается в части 6.

COM и CLR-среда основаны на различных компонентных архитектурах, и все же Microsoft позаботилась о механизмах «мирного сосуществования» этих технологий. Обычно удается без проблем обеспечить совместимость COM и CLR. Маловероятно, что в мире .NET вы станете использовать COM в качестве компонентной архитектуры, однако вряд ли откажетесь от ATL (Active Template Library) Server — высокопроизводительного инструмента создания Web-сайтов.

В этом издании мы «освежили» описания ATL и MFC, так как они все еще остаются действенными инструментами разработки. Более того, вы узнаете, как использовать уже созданный код при переходе на .NET.

Управляемый C++ и C#

Вместе с платформой .NET компания Microsoft представила новый язык — C# (произносится: «си шарп»). Он похож на C++, и в нем такой же основанный на фигурных скобках синтаксис. В C# сняты проблемы, характерные для C++ (в частности, управление указателям), но сохранены многие достоинства C++ (в частности, виртуальные функции). Кстати, компилятор C# генерирует управляемый код, способный работать в CLR-среде.

Понятно, что весь мир не перейдет на C# в один день — слишком много кода написано на C++, да и разработчикам потребуется время, чтобы привыкнуть к C#. Помимо прочего, в .NET представлены новые дополнения к C++ (Managed Extensions for C++), или управляемый C++, программы на котором исполняет CLR-среда. Управляемый C++ позволит быстро модернизировать унаследованный код на C++ для работы в CLR-среде. Преобразование «обычного» кода на C++ в управляемый оз-

начает щедрое его «сдабривание» определенными ключевыми словами. В конечном итоге по завершении работы компилятора тексты на C# и на управляемом C++ преобразуются в одинаковый исполняемый код. В мире .NET вы скорее всего будете создавать новые программы на C#, а управляемый C++ применять для преобразования своего старого кода в совместимый с .NET.

.NET и платформа Java

В последние годы вырос интерес к языку программирования Java и платформе разработки на Java, которые Интернет-разработчики полюбили за средства распространения пользовательского интерфейса (с помощью Java-апплетов) и поддержку разработки корпоративных приложений средствами Java Enterprise Edition. Теперь лучшей платформой для разработки Интернет-приложений стала платформа .NET. В отличие от Java, где требуется писать код только на этом языке, .NET позволяет для получения одинакового исполняемого кода применять разные языки. Для своих программ вы вправе использовать «обычный» и управляемый C++ (им посвящена настоящая книга), Visual Basic .NET, C# или любой другой язык, поддерживаемый .NET. Исходный текст преобразуется в код на промежуточном языке, который во время исполнения трансформируется в машинные команды. В период выполнения .NET управляет исполнением кода, обеспечивая такие преимущества, как сборка мусора и повышенная безопасность кода.

Кому адресована эта книга

Visual C++ .NET с ее мощными средствами разработки приложений и поддержкой .NET предназначена для профессиональных программистов, и эта книга — тоже для них. Мы считаем, что вы достаточно хорошо знаете C++ и можете написать оператор *if*, не обращаясь к справочнику. Мы также предполагаем, что у вас есть некоторый опыт работы на C++ — по крайней мере вы прошли некоторый курс обучения или прочитали какую-нибудь книгу о нем, хотя, может быть, и не писали очень больших программ. Изучение C++ можно сравнить с изучением французского. Даже если вы учили его в школе, вы не сможете свободно говорить на нем, пока не поедете во Францию и не пообщаетесь с носителями языка.

Мастера Visual C++ экономят время и повышают корректность кода, но программист должен понимать, что за код они генерируют, и — что самое важное — разбираться в структуре библиотек MFC и ATL и внутренних механизмах работы Windows и .NET. Однако мы не считаем, что вы знакомы с программированием для Windows и .NET. Мы уверены, что опытные программисты на C++ могут изучать работу с Windows как посредством MFC, так и .NET. Знание C++ важнее знания API Win32. Тем не менее мы полагаем, что читатель умеет работать с Windows и Windows-приложениями.

Что, если у вас уже есть опыт работы с API Win32 или с библиотекой MFC? В этой книге найдется кое-что интересное и для вас. Вы узнаете о новых возможностях, таких как интерфейс со многими окнами верхнего уровня (Multiple Top-Level Interface, MTI) и мастера Visual C++ .NET. Если вы еще не знакомы с моделью COM (Component Object Model), эта книга содержит некоторые важные теоретические сведения, с которых начнется ваше знакомство с элементами управления ActiveX.

Вы также познакомитесь с ATL Server и шаблонами OLE DB. Мы обсудим программирование на C++ для Интернета (в том числе Dynamic HTML). Наконец, мы расскажем о некоторых скрытых особенностях управляемого C++.

Что не вошло в книгу

В одной книге невозможно рассмотреть все стороны программирования для Windows и .NET. Мы исключили темы, требующие специализированных аппаратных или программных средств, такие как MAPI, TAPI и работа с коммуникационными портами. Мы рассмотрим элементы управления ActiveX и написание ActiveX-элементов с помощью ATL, но оставим подробный рассказ об ActiveX Адаму Деннигу (Adam Dennig) и его книге «ActiveX Controls Inside Out» (Microsoft Press, 1997). Мы познакомим вас с основами управления памятью в 32-разрядных системах, теории DLL и приемами многопоточного программирования, но для серьезно изучения этих тем вам потребуется книга Джеффри Рихтера (Jeffrey Richter) «Programming Applications for Microsoft Windows» (Microsoft Press, 1997) (Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. М.: «Русская Редакция» и «Питер», 2001). Другая полезная книга — «MFC Internals» Джорджа Шеферда (George Shepherd) и Скотта Уингоу (Scot Wingo) (Addison-Wesley, 1996). Мы расскажем об основах .NET, детали программирования для этой платформы см. в книге Джеффри Рихтера «Applied Microsoft .NET Framework Programming» (Microsoft Press, 2002) (Рихтер Дж. Программирование на платформе MS .NET FRAMEWORK. М.: «Русская Редакция», 2002).

Как пользоваться книгой

В начале работы с Visual C++ .NET книгу лучше читать последовательно. Затем ее можно использовать как справочник. Из-за тесной взаимосвязи между большинством компонентов каркаса приложений невозможно посвятить каждому аспекту отдельную главу, поэтому книгу, безусловно, нельзя рассматривать как энциклопедию. Читая ее, полезно иметь под рукой интерактивную справочную систему Visual C++ .NET для просмотра информации о классах и функциях-членах.

Если вы имеете опыт работы с предыдущими версиями Visual C++, просмотрите часть 1, чтобы получить представление о новых возможностях. Затем пропустите первые основы MFC в части 2, но прочитайте главы, где обсуждаются детали каркаса. Обязательно прочитайте часть 6, посвященную .NET. Большинство программистов движется именно в этом направлении, а Visual C++ .NET полностью поддерживает модель программирования в .NET.

Структура книги

Как видно из оглавления, книга состоит из шести частей и двух приложений.

Часть1: Основные сведения о Windows, Visual C++ .NET и каркасе разработки приложений

В этой части мы стремились поддерживать баланс между теорией и практикой. После беглого обзора Win32 и компонентов Visual C++ .NET вы кратко ознакоми-

тесь с каркасом приложения MFC и архитектурой «документ-вид». При помощи классов библиотеки MFC вы создадите простую программу «Hello, world!», насчитывающую всего 30 строк текста.

Часть 2: Основы MFC

В документации по MFC все элементы каркаса приложения рассмотрены последовательно в предположении, что вы знакомы с оригинальным API Windows. Вторая часть книги посвящена только одному важному компоненту каркаса приложения — *виду*, или *представлению* (view), которое на самом деле является окном. Здесь вы узнаете то, что и так известно опытному программисту для Windows, но здесь эта тема представлена в контексте C++ и классов библиотеки MFC. Инструменты Visual C++ .NET позволят нам избавиться от большей части рутинного кодирования, с которым приходилось мириться программистам на «заре» Windows.

Во этой части затронуто много тем, в том числе программирование графики с использованием растровых изображений, обмен данными в диалоговом окне, использование элементов управления ActiveX, 32-разрядное управление памятью и многопоточное программирование. Упражнения помогут написать довольно сложные программы для Windows, не обращаясь к расширенным средствам каркаса приложения.

Часть 3: Архитектура «документ-вид» в MFC

В этой части рассматривается «настоящее» программирования с использованием MFC — в архитектуре «документ-вид». Вы узнаете, что такое *документ* (нечто более абстрактное, чем документ в текстовом процессоре) и как подключить его к представлению, с которым вы познакомились в части 2. Написав класс документа, вы будете поражены, насколько библиотека MFC упрощает программирование файлового ввода/вывода и вывода на печать.

Вы также узнаете об обработке командных сообщений, панелях инструментов и состояния, разделяемых окнах-рамках и контекстно-зависимой справке. Здесь также рассмотрены виды интерфейсов в Windows-приложениях: однодокументный (Single Document Interface, SDI), многодокументный (Multiple Document Interface, MDI) и интерфейс на основе многих окон верхнего уровня (Multiple Top-Level Windows Interface, MTI). Последний является стандартом интерфейса современных Windows-приложений, например Microsoft Word.

Здесь же обсуждается, как с помощью MFC написать динамически подключаемые библиотеки (Dynamic Link Libraries, DLL). Вы поймете различие между обычной DLL и DLL-расширением.

Часть 4: COM, Automation, ActiveX и OLE

COM заслуживает нескольких книг. Здесь вы познакомитесь с основами теории COM с точки зрения MFC. Затем речь пойдет об Automation — связующем звене между C++ и VBA (Visual Basic for Applications). Вы познакомитесь с унифицированным обменом данными (Uniform Data Transfer) и основами составных документов и внедренных объектов. В этой части также рассмотрена поддержка OLE DB в классах библиотеки ATL.

Часть 5: Создание приложений для Интернета

Эта часть начинается с технического введения в Интернет, в котором рассматриваются протокол TCP/IP и интерфейсы программирования для Интернета. Вы узнаете, как создавать серверы средствами ATL Server и как программировать с использованием Dynamic HTML.

Часть 6: .NET и дальше

Разработка приложений для Интернета больше не ограничивается созданием Web-сайтов, предназначенных для простого просмотра, — нужно создавать Web-сайты, доступ к которым возможен из программ. Все базовые средства связи уже существовали, но до появления XML не удавалось достичь согласия относительно передачи запросов методов через Интернет. Платформа .NET является прорывом по крайней мере в двух отношениях: в области Web-сервисов и обслуживаемого сервером пользовательского интерфейса. Она обеспечивает полную поддержку этих технологий и предоставляет новый метод создания клиентских пользовательских интерфейсов — на основе Windows Forms. В этой части объясняется, что собой представляет платформа .NET и что вы можете делать с ее помощью. Здесь есть главы, посвященные CLR-среде и управляемому коду, а также программированию управляемых компонентов средствами ASP.NET и ADO.NET.

Приложения

Приложение А «Функции карты сообщений в библиотеке MFC» содержит список макросов карты сообщений и прототипов соответствующих им функций-обработчиков. Обычно код таблицы автоматически генерируется мастерами, доступными в окне Class View, но иногда его приходится вводить вручную.

В приложении Б «Идентификация MFC-классов во время выполнения и динамическое создание объектов» приведено описание системы информации о классах периода выполнения и динамического создания объектов MFC. Эта система существует независимо от RTTI (runtime type information) — средства, описанного в стандарте ANSI C++.

Win32 или Win16?

Windows 3.1 все еще встречается на некоторых старых компьютерах. Однако нет особого смысла тратить средства и время на создание новых программ для устаревших технологий. Это, 6-е, издание книги посвящено 32-разрядному программированию для Microsoft Windows 98/Me и Windows NT/2000/XP с использованием API Win32. Если вам действительно нужно писать 16-разрядные программы, отыщите второе издание.

Требования к системе

Для работы с этой книгой требуется установленная копия Visual C++ .NET или Visual Studio .NET. Любой компьютер, удовлетворяющий минимальным требованиям для установки Visual C++ .NET, подойдет для работы с примерами. Заметим, что в ОС Windows XP Home Edition и Windows NT 4.0 нельзя разместить Web-приложе-

ния ASP.NET на каркасе .NET Framework. Такие приложения можно собирать на указанных системах, но развертывать их придется на более подходящей ОС.

Примеры программ на прилагаемом компакт-диске

Компакт-диск содержит исходные тексты всех примеров программ, а также некоторые вспомогательные материалы. Чтобы работать с файлами на компакт-диске, вставьте его в дисковод и выберите нужную команду из меню появившегося окна. Если автозапуск компакт-дисков на компьютере отключен, окно с меню не откроется, и вам придется запустить файл StartCD.exe в корневом каталоге диска. Для установки файлов с примерами требуется примерно 60 Мб свободного пространства на жестком диске. При выполнении программ-примеров настоятельно рекомендуем сверяться с текстом книги.

Об обычной программе на С, использующей API Windows, все скажут ее исходные тексты. При использовании каркаса приложения библиотеки MFC не все так просто. Значительную часть кода С++ генерирует мастер MFC Application Wizard, а ресурсы порождаются соответствующими редакторами ресурсов. Примеры в начальных главах содержат пошаговые инструкции по использованию указанных инструментов для генерации и редактирования исходных файлов — вводить код вручную практически не придется. Для примеров из глав середины книги используйте тексты с компакт-диска, но просмотрите последовательность шагов создания программы, чтобы понять роль редакторов ресурсов и мастеров. В последних главах исходные тексты примеров приведены не полностью. При изучении этих примеров потребуется обратиться к примерам с компакт-диска.

Кроме файлов примеров, на компакт-диске хранятся две электронные версии книги: автономная и интегрируемая в справочную систему Visual Studio.

Дополнительные библиотеки Windows Forms

Одна из самых привлекательных особенностей, «продававшая» MFC на протяжении 90-х, — библиотеки классов для расширения каркаса приложений. С появлением Windows Forms приходится следить за выходящими библиотеками-расширениями.

MFC и ее расширения ограничены языком С++, однако CLR-среда в .NET позволяет использовать самые разнообразные синтаксисы для написания приложений на основе Windows Forms, в том числе C#, Visual Basic .NET и управляемый С++. Компания Syncfusion из города Кэри (Северная Каролина) создала набор инструментов, облегчающих программирование для .NET. Набор Essential Suite содержит компоненты, которые позволят сделать ваши приложения Windows Forms более надежными и совершенными. Полнофункциональную 15-дневную пробную версию Essential Suite, а также интерактивное приложение Interactive Showcase, демонстрирующее некоторые компоненты Syncfusion в действии, можно скачать с сайта *http://www.syncfusion.com*. Компоненты работают в CLR-среде, поэтому их можно использовать в программах на управляемом С++, а также на C# и Visual Basic .NET.

Поддержка

Мы стремились избежать ошибок в книге и на прилагаемом к ней компакт-диске. Исправления к данной книге Microsoft Press предоставляет по адресу: *http://www.microsoft.com/mspress/support/*.

База знаний Microsoft Press Knowledge Base доступна напрямую на Web-странице *http://www.microsoft.com/mspress/support/search.asp*.

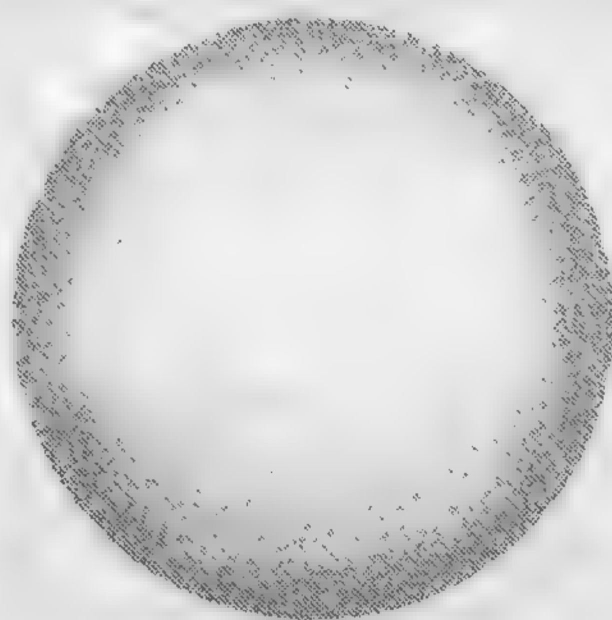
Если у вас есть комментарии или вопросы, ответов на которые не удалось найти в базе Microsoft Press Knowledge Base, направляйте их в Microsoft Press по обычной или электронной почте:

Microsoft Press
Attn: Programming with Microsoft Visual C++ .NET Editor
One Microsoft Way
Redmond. WA 98052-6399
MSPINPUT@MICROSOFT.COM

Техническая поддержка по указанным почтовым адресам не предоставляется. Подробную информацию о технической поддержке конкретных программных продуктов Microsoft см. на Web-узле компании по адресу *http://support.microsoft.com*.

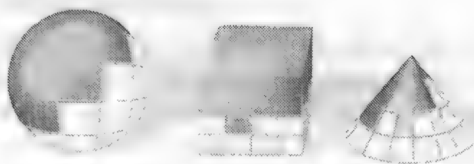
ЧАСТЬ I

ОСНОВНЫЕ СВЕДЕНИЯ О WINDOWS, VISUAL C++ .NET И КАРКАСЕ РАЗРАБОТКИ ПРИЛОЖЕНИЙ



ГЛАВА

1



Microsoft Windows и Visual C++ .NET

В начале 90-х годов борьба шла за «настольные» операционные системы, и на сегодняшний день она завершилась — Microsoft Windows царствует на подавляющем большинстве персональных компьютеров. В этой главе мы обсудим низкоуровневую модель программирования в Windows (и в Win32, в частности) и увидим, как взаимодействуют компоненты Visual C++ .NET при создании прикладной программы. Попутно вы узнаете кое-что новое и о Windows.

Модель программирования в Windows

Программирование в Windows отличается от старомодного стиля, ориентированного на обработку последовательностей команд или запросов. Давайте разберемся в основах самой Windows. За точку отсчета возьмем хорошо известную MS-DOS. Даже если вы сейчас и не пишете для MS-DOS, такая модель программирования вам скорее всего знакома.

Обработка сообщений

В MS-DOS-программе на С обязательно имеется функция *main*. ОС вызывает ее при запуске программы, и с этого момента вы по сути можете делать что угодно. Если программе надо узнать, какие клавиши нажаты на клавиатуре, или как-то иначе задействовать сервисы ОС, она вызывает соответствующие функции, например *getchar*, или обращается к более специализированной библиотеке функций, оперирующих с символами.

В Windows же система при запуске программы вызывает функцию *WinMain*. В любом приложении должна присутствовать эта функция, на которую возлагается ряд специфических задач. И важнейшая — создание основного окна програм-

мы, у которого должен быть свой код обработки сообщений, поступающих от ОС. Существенное различие между MS-DOS- и Windows-программой состоит в способе получения введенных пользователем данных: первая обращается прямо к ОС, а второй нужны поступающие от ОС сообщения.

Примечание Многие среды разработки для Windows, включая Microsoft Visual C++ .NET с библиотекой классов Microsoft Foundation Class (MFC) Library 7.0, упрощают программирование, скрывая функцию *WinMain* и структурируя процесс обработки сообщений. Библиотека MFC позволяет обойтись без написания функции *WinMain*, но важно понимать, как взаимодействуют ОС и ваша программа.

Большинство сообщений в Windows строго определено и относится ко всем программам. Так, сообщение *WM_CREATE* передается при создании окна, *WM_LBUTTONDOWN* — при нажатии левой кнопки мыши, *WM_CHAR* — при вводе символа, а *WM_CLOSE* — при закрытии окна пользователем. У всех сообщений есть два 32-разрядных параметра, передающих такие сведения, как координаты курсора, код нажатой клавиши и т. п. Сообщения группы *WM_COMMAND* отправляются соответствующему окну в ответ на выбор пользователем команд в меню, щелчки кнопок диалоговых окон и т. п. Параметры командных сообщений зависят от структуры меню конкретного окна. Кроме предопределенных сообщений, программист вправе определять свои, так называемые *пользовательские сообщения*, которые позволено направлять в любое окно. Из-за возможности создания таких сообщений C++ становится слегка похожим на Smalltalk.

Не ломайте пока голову над тем, как «увязываются» сообщения и программный код. За это отвечает *каркас приложения* (application framework). Однако обработка Windows-сообщений накладывает на структуру программ весьма жесткие ограничения. Не пытайтесь выстраивать программы для Windows по аналогии с MS-DOS. Изучите примеры этой книги и приготовьтесь начать все сначала.

Интерфейс графического устройства

Многие программы MS-DOS записывали данные прямо в видеопамять и порт принтера. Недостаток этого метода в том, что разработчику приходилось создавать отдельные драйверы для каждой из множества моделей видеоплат и принтеров. В Windows предусмотрен особый слой абстрагирования от оборудования — *интерфейс графического устройства* (Graphics Device Interface, GDI). Драйверы видеокарт и принтеров предоставляет сама Windows, благодаря чему программе не надо «знать», какие видеоплата и принтер подключены к системе. Вместо обращения к оборудованию, программа вызывает GDI-функции, ссылающиеся на определенную структуру данных — *контекст устройства* (device context). Windows сопоставляет структуру контекста устройства физическому устройству и выдает соответствующие команды ввода/вывода. GDI обеспечивает почти такую же скорость, как и прямой доступ к видеопамяти, и позволяет нескольким Windows-программам одновременно работать с дисплеем.

Чуть далее мы познакомимся с GDI+. Как вы, наверное, догадались, это следующая версия GDI. Сервисы GDI+ предоставляются через определенный набор

классов C++ на управляемом языке, т. е. как код, выполняемый общеязыковой средой исполнения (Common Language Runtime, CLR). В GDI+ есть ряд функций, которых нет в «классическом» GDI, в том числе градиентные кисти, кардинальные сплайны, независимые объекты-пути, масштабируемые области, альфа-сопряжение и поддержка множества форматов изображений.

Программирование, ориентированное на ресурсы

В MS-DOS вы определяли данные при помощи инициализирующих констант либо считывания из специальных файлов. В Windows же данные размещаются в файле ресурсов, представляемых в нескольких стандартных форматах. При генерации исполняемой программы *компоновщик* (linker) комбинирует двоичные файлы ресурсов с кодом, полученным на выходе компилятора C++. Файлы ресурсов содержат растровые изображения, или *битовые карты* (bitmaps), *значки* (icons), определения меню, описания структуры диалоговых окон и строки. Они могут хранить даже описания особых форматов ресурсов, определенных программистом.

Программа редактируется в текстовом редакторе, но ресурсы обычно редактируют, применяя инструменты, обеспечивающие режим WYSIWYG (What You See Is What You Get — «что видишь, то и получишь»). Так, при разметке структуры диалогового окна отдельные ее элементы выбирают из набора значков — *палитры элементов управления* (control palette) — нужные элементы (кнопки, списки и пр.) и с помощью мыши размещают на форме и подгоняют по размеру. Графический редактор Visual C++ .NET позволяет эффективно редактировать ресурсы стандартных форматов.

Управление памятью

В каждой последующей версии Windows управление памятью становится проще. Если вы уже наслушались леденящих душу историй о *блокирующих описателях памяти* (handles), *переходах* (thunks) и прочих ужасах, не паникуйте: все это в прошлом. Сегодня вы просто выделяете память, а об остальном заботится Windows. Современные способы управления памятью в Win32 (в частности, виртуальную память и проецируемые в память файлы) мы обсудим в главе 10.

Динамически подключаемые библиотеки

В среде MS-DOS все объектные модули программы связываются статически на стадии компоновки. Windows разрешает динамическое связывание, т. е. загрузку и подключение к программе специальных библиотек в период исполнения. К одной и той же *динамически подключаемой библиотеке* (Dynamic-Link Library, DLL) могут обращаться сразу несколько программ, что экономит память и дисковое пространство. Кроме того, динамическое подключение позволяет создавать модульные программы, так как DLL-модули компилируются и тестируются отдельно.

Изначально DLL разрабатывались в расчете на язык C, а C++ привнес в этот процесс кое-какие сложности. Впрочем, разработчики MFC сумели скомпоновать все классы каркаса приложений в несколько законченных DLL. Это значит, что упомянутые классы можно подключать к приложению как статически, так и ди-

намически. Кроме того, вы вправе создавать свои DLL-модули расширения, построенные на основе DLL-модулей библиотеки MFC. Обо всем этом см. главу 22.

Интерфейс прикладного программирования Win32

На заре Windows программисты писали приложения на C в расчете на интерфейс прикладного программирования (Application Programming Interface, API) Win16. Сейчас для Win16 пишут немногие — большинство работает с Win32. Основное различие между Win16-функциями и их Win32-эквивалентами — в расширении представления параметров с 16 до 32 разрядов. Поэтому, несмотря на постоянную эволюцию Windows API, программисты оказались в значительной степени изолированы от различий в API, так как они создают приложения в соответствии со стандартом MFC, поддерживающим как Win16, так и Win32.

Компоненты Visual C++ .NET

Microsoft Visual C++ .NET объединяет две законченные системы разработки Windows-приложений. Можно создавать Windows-программы на языке C, используя только Win32 API. Программирование в Win32 на C описано в книге Чарльза Петцольда (Charles Petzold) «Programming Windows» (Microsoft Press, 1996). [В Microsoft Press вышла его новая книга — «Programming Microsoft Windows with C#» («Программирование для MS Windows на C#», «Русская Редакция», М., 2002 г.), — где рассказывается о программировании с применением C# и Windows Forms. Программирование в Windows с использованием Windows Forms и C++ мы тоже обсудим.] Заметно облегчают работу по низкоуровневому Win32-программированию многие инструменты Visual C++ .NET, в том числе редакторы ресурсов. Для ускорения разработки Windows-программ можно воспользоваться и библиотеками каркаса приложений, такими как MFC и Windows Forms.

Наконец, Visual C++ .NET содержит библиотеку шаблонов ActiveX (ActiveX Template Library, ATL), позволяющую разрабатывать элементы управления ActiveX. Программирование в ATL не сводится к программированию ни в Win32, ни в MFC — оно очень сложно и заслуживает отдельной книги. И все же мы поговорим об ATL-разработке, которая больше подходит для программирования высокопроизводительных компонентов в среде Web-серверов.

Эта книга — о программировании на C++ с использованием каркаса приложений — библиотеки MFC, составляющей часть Visual C++ .NET. Вы будете применять классы C++, описанные в документах библиотеки Microsoft Foundation Class Library Reference, а также специальные инструменты Visual C++ .NET для библиотеки MFC, в частности, MFC Application Wizard и Class View.

Примечание Работа с MFC-библиотекой не лишает программиста возможности применять функции Win32. На самом деле в программах на основе этой библиотеки почти всегда придется напрямую вызывать Win32-функции.

Прежде чем вы возьметесь за свое первое приложение, мы вкратце опишем компоненты Visual C++ .NET — это поможет вам понять, с чем придется иметь дело. Рис. 1-1 иллюстрирует процесс создания программ в Visual C++ .NET.

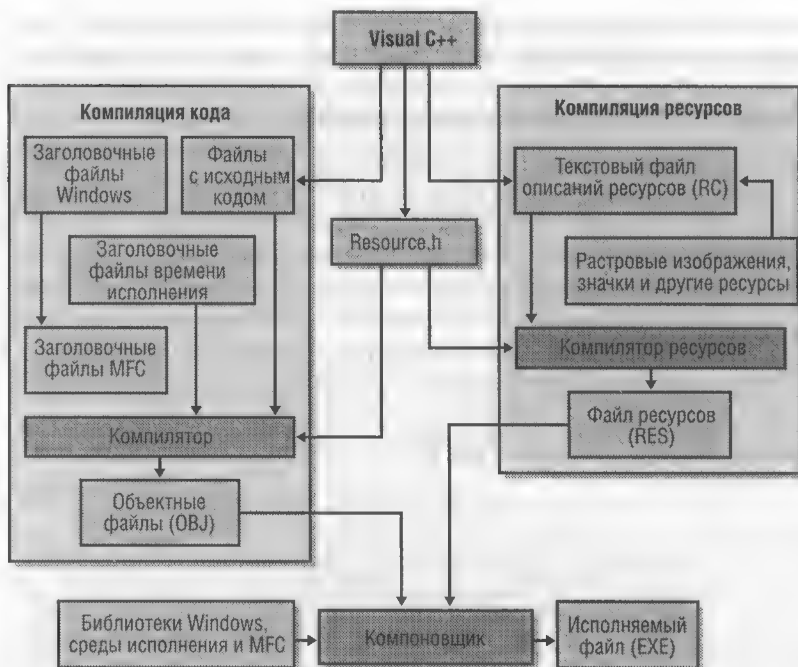


Рис. 1-1. *Создание программы в Visual C++ .NET с применением MFC*

Microsoft Visual C++ .NET и процесс сборки программ

Visual C++ .NET является частью Visual Studio .NET — комплекта средств разработки. *Интегрированную среду разработки* (Integrated Development Environment, IDE) Visual C++ .NET используют и другие средства разработки, например Microsoft J++. Эта среда начала свой долгий путь с Visual Workbench — среды, основанной на QuickC for Windows. Теперь в ней есть *стыкуемые окна* (docking windows), *конфигурируемые панели инструментов* (configurable toolbars) и *настраиваемый редактор* (customizable editor), способный исполнять макросы. Встроенная справочная система, интегрированная с программой просмотра библиотеки Microsoft Developer Network (MSDN), теперь работает по принципу Web-браузера (рис. 1-2).

Если вы работали с прежними версиями Visual C++ .NET, то понимаете, как функционирует Visual C++ .NET (хотя некоторые меню и поменялись). Но если интегрированные среды вам в новинку, сначала надо уяснить, что такое *проект* (project). Проект — это набор взаимосвязанных исходных файлов, компиляция и компоновка которых позволяет создать исполняемую Windows-программу или DLL. Исходные файлы проекта обычно хранятся в отдельном подкаталоге. Кроме того, зачастую проект зависит от многих файлов, расположенных вне подкаталога проекта, таких как *включаемые* (include) и библиотечные файлы.

Visual Studio .NET также поддерживает разработку приложений вне интегрированной среды с применением *сборочных файлов проекта*, или make-файлов (make files). Make-файл содержит параметры компилятора и компоновщика, а также отражает все взаимосвязи между исходными файлами. То есть вы вправе создать

make-файл вручную и исполнять его средствами программы NMAKE.EXE. (Файлу с исходным кодом нужны включаемые файлы, исполняемому файлу — определенные объектные модули, библиотеки и т. д.) Программа NMAKE.EXE считывает make-файл, а затем вызывает компилятор, ассемблер, компоновщик и компилятор ресурсов, чтобы получить на выходе нужный файл — исполняемый или библиотечный. Эта программа, руководствуясь указанной ей информацией, заставляет, например, компилятор генерировать OBJ-файл из определенного CPP-файла. Кстати, в Visual C++ .NET больше недоступна возможность экспорта make-файла активного проекта из среды разработки. Для сборки проектов Visual C++ .NET в командной строке применяется параметр компилятора *Devenv*.

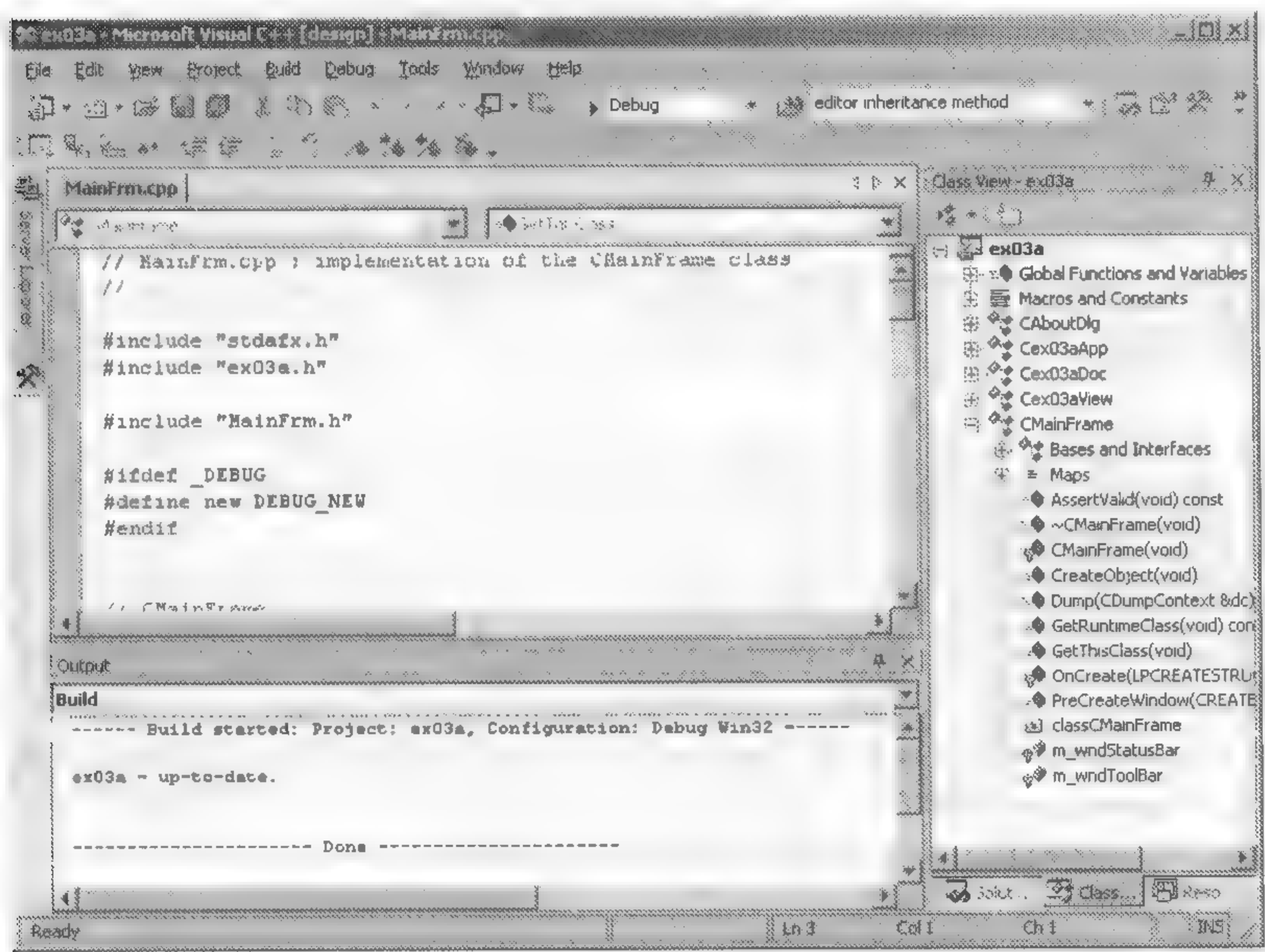


Рис. 1-2. *Окна Visual C++ .NET*

В проекте Visual C++ .NET взаимозависимости между отдельными частями описаны в текстовом файле проекта с расширением VCPROJ. А специальный текстовый файл решения с расширением SLN содержит список всех проектов данного решения. *Решение* (Solution) объединяет несколько проектов (но во всех примерах этой книги оно состоит из одного проекта). Чтобы начать работу с существующим проектом, достаточно открыть в Visual C++ .NET соответствующий SLN-файл, и проект можно редактировать и собирать.

Visual C++ .NET также создает промежуточные файлы нескольких типов (табл. 1-1).

Табл. 1-1. Типы файлов, создаваемых в проектах Visual C++ .NET

Расширение файла	Описание
APS	Поддержка просмотра ресурсов
BSC	Информация браузера
IDL	Файл на языке описания интерфейсов IDL
NCB	Поддержка просмотра классов

см. след. стр.

Табл. 1-1. (*продолжение*)

Расширение файла	Описание
SLN	Файл решения. *Не удаляйте и не изменяйте эти файлы средствами текстового редактора!*
SUO	Поддержка параметров и конфигурации решения
VCPROJ	Файл проекта. *Не удаляйте и не изменяйте эти файлы средствами текстового редактора!*

Редакторы ресурсов и просмотр ресурсов проекта

Открыв окно просмотра ресурсов Resource View (команда Resource меню View) в Visual C++ .NET, можно редактировать ресурсы проекта. В основном окне располагается *редактор ресурсов* (resource editor), предназначенный для конкретного типа ресурсов. Это может быть и WYSIWYG-редактор меню, и мощный графический редактор диалоговых окон, и набор инструментов для операций со значками, растровыми изображениями и строками. Кроме стандартных элементов управления Windows, редактор диалоговых окон позволяет вставлять и элементы управления ActiveX.

В каждом проекте обычно есть один RC-файл, в котором описаны ресурсы меню, диалоговых окон, строк и «быстрых клавиш». Этот файл также обычно содержит операторы *#include*, «собирающие» ресурсы из других подкаталогов. В эти ресурсы включаются как характерные для конкретного проекта, например, BMP-файлы или файлы значков (ICO), так и общие для всех программ Visual C++ .NET, например, строки сообщений об ошибках. Модифицировать RC-файлы вне графического редактора не рекомендуется. Редактор ресурсов способен обрабатывать EXE- и DLL-файлы, благодаря чему вы, например, сможете заимствовать растровые изображения и значки у «чужих» Windows-программ.

Компилятор C/C++

Компилятор Visual C++ .NET способен обрабатывать исходный код и на С, и на С++ — язык он определяет по расширению файла с исходным кодом. Расширение С указывает на код на С, а СРР или СХХ — на С++. Компилятор удовлетворяет стандартам ANSI, включая самые последние рекомендации рабочей группы по библиотекам С++, и обладает рядом расширений, введенных Microsoft. Шаблоны, обработка исключений и *идентификация типов во время исполнения* (runtime type identification, RTTI) — все это поддерживается в Visual C++ .NET. Введена также (хотя и не интегрирована в библиотеку MFC) *стандартная библиотека шаблонов* (Standard Template Library, STL).

Редактор исходного текста

Visual C++ .NET содержит мощный редактор исходного текста, поддерживающий массу возможностей: выделение цветом синтаксических конструкций, автоотступы, раскладки клавиатурных команд популярных редакторов (например, VI и EMACS), а также высококачественную печать. В версии 6.0 Visual C++ появилась

впечатляющая новая функция AutoComplete. Если вы работали с приложениями семейства Microsoft Office или с Microsoft Visual Basic, то наверняка знаете, что это такое. Когда вы набираете несколько начальных символов оператора программы, редактор предлагает выбрать ключевое слово из списка вариантов. Это очень удобно, если вы работаете с объектами C++ и забыли точное имя члена класса, неважно, функции или переменной, — все они в списке перед вами. Благодаря AutoComplete не нужно забивать память форматами тысяч функций Win32 API или постоянно обращаться к интерактивной справочной системе.

Компилятор ресурсов

Компилятор ресурсов в Visual C++ .NET считывает созданный в графическом редакторе ASCII-файл описания ресурсов (RC) и создает для компоновщика двоичный RES-файл.

Компоновщик

Компоновщик считывает OBJ- и RES-файлы, сгенерированные компилятором C/C++ и компилятором ресурсов, и обращается к LIB-файлам за MFC-кодом, кодом библиотек периода исполнения и Windows. После этого он создает EXE-файл проекта. Возможность *компоновки с приращением* (incremental link) заметно ускоряет процесс, когда в исходные файлы вносятся лишь незначительные изменения. Заголовочные файлы MFC содержат операторы *#pragma* (специальные директивы компилятора), указывающие нужные библиотечные файлы, так что явным образом сообщать компоновщику, к каким библиотекам обращаться, не надо.

Отладчик

Если ваши программы начинают работать сразу, отладчик не нужен. Ну а нам, простым смертным, приходится им от случая к случаю пользоваться. Интегрированный в Visual C++ .NET отладчик (рис. 1-3) обладает как возможностями отладчиков более ранних версий Visual C++ и Visual Basic, так и новыми. Вот они.

- **Отладка программ на нескольких языках** Visual Studio .NET позволяет отлаживать решения, отдельные проекты которых написаны на разных языках.
- **Подключение к работающей программе** Можно подключаться и отлаживать программу, которая уже исполняется вне среды разработки Visual Studio .NET.
- **Удаленная отладка** Теперь вы вправе подключиться и отладить программу на другом компьютере.
- **Отладка Web-приложений ASP.NET** Файлы ASP.NET компилируются, поэтому к их отладке следует относиться так же, как и к отладке программ на других языках. За счет этого значительно облегчается отладка Web-приложений.
- **Классы .NET Framework для отладки и трассировки кода** Упрощают размещение операторов трассировки кода в тексте программы. Эти классы состоят из управляемого кода, поэтому их можно исполнять в рамках управляемого C++.

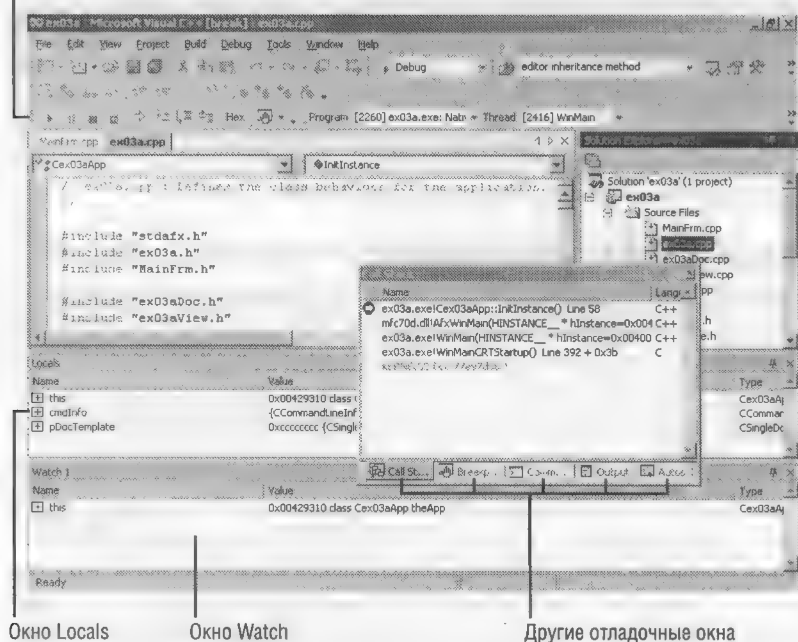


Рис. 1-3. *Окно отладчика Visual C++ .NET*

В окнах Variables и Watch можно развернуть указатели объектов, что приведет к отображению всех переменных-членов производного класса и базовых классов. Если же поместить указатель мыши на простую переменную, отладчик покажет ее значение в маленьком окошке. При отладке программу надо собирать с параметрами компилятора и компоновщика, которые предусматривают генерацию специальной отладочной информации.

В Visual C++ .NET есть функция Edit and Continue, позволяющая прерывать процесс отладки, изменять текст программы, а затем продолжать отладку уже измененной программы. Это сильно ускоряет отладку, так как больше не нужно вручную выходить из отладчика, перекомпилировать программу и лишь затем снова запускать отладку. Чтобы задействовать эту функцию, достаточно в процессе отладки отредактировать исходный код и нажать кнопку Continue (синий треугольник на панели инструментов). Visual C++ .NET скомпилируют измененную программу и перезапустит отладчик.

MFC Application Wizard

MFC Application Wizard — генератор кода, создающий рабочую заготовку Windows-приложения с теми компонентами, именами классов и исходными файлами, которые вы задали в его диалоговых окнах. Изучая примеры этой книги, вы будете интенсивно использовать мастер MFC Application Wizard. Только не путайте его со старыми генераторами кода, формировавшими весь код приложения. MFC Application Wizard создает минимум — главная функциональность сосредоточена в базовых классах каркаса приложений.

MFC Application Wizard поможет вам побыстрее приступить к работе над новым приложением. Более того, мастера расширяемы, т. е. вы вправе создавать собственные генераторы кода. И если ваша команда занята разработкой множества проектов, скажем, в области телекоммуникаций, можно построить свой мастер для автоматизации работы.

Окно Class View

Окно Class View открывается при выборе команды Class View в меню View и отображает дерево всех классов проекта с функциями-членами и переменными-членами (рис. 1-2). Чтобы увидеть код элемента, его нужно дважды щелкнуть. При внесении изменений в исходный текст содержимое окна Class View автоматически обновляется. В предыдущих версиях Visual C++ для выполнения практически всех задач по управлению кодом классов применялся один-единственный компонент — ClassWizard. Ему на смену пришли несколько новых мастеров, отвечающих за выполнение отдельных задач, таких как добавление совсем новых классов, виртуальных функций в классы и функций-обработчиков сообщений. В частности, на смену добавлению классов и функций пришла утилита Class View.

Средство просмотра исходного кода

Если вы пишете программу «с нуля», то, видимо, хорошо представляете себе ее структуру: все файлы с исходным кодом, классы и функции-члены. Но чтобы разобраться в чужой программе, вам точно потребуется помощь. В Visual C++ .NET предусмотрено *средство просмотра исходного кода* (Source Browser), или попросту средство просмотра. Оно позволяет анализировать (и редактировать) программу через призму конкретного класса или функции, а не файла или файлов. В чем-то оно напоминает инструменты-инспекторы, доступные с другими объектно-ориентированными библиотеками, например Smalltalk. Средство просмотра работает в следующих режимах.

- **Definitions and References (определения и ссылки)** Выбрав функцию, переменную, тип, макрос или класс, вы увидите, где они определены и где именно используются.
- **Call Graph/Caller Graph (схема вызываемых или вызывающих функций)** Выбрав функцию, вы увидите графическое представление функций, вызываемых ею или вызывающих ее.
- **Derived Class Graph/Base Class Graph (схема производных или базовых классов)** Графическая схема иерархии классов. Выбрав класс, вы увидите его производные или базовые классы. При помощи мыши можно управлять степенью детализации («развертки») иерархии.
- **File Outline (конспект файла)** Для выбранного файла классы, члены классов и функции появляются вместе с указанием на все места программы, где они определены и используются.

Типичный пример окна средства просмотра см. в главе 3.

Примечание Если вы перегруппируете строки в любом из файлов с исходным кодом, Visual C++ .NET обновит используемую средством просмотра базу данных в момент повторной сборки проекта. На это, естественно, уйдет дополнительное время.

Кроме упомянутого выше средства, в Visual C++ .NET имеется новый режим просмотра — Class View, независимый от базы данных, используемой при просмотре. В этом режиме показывается дерево всех классов в проекте вместе с функциями-членами и переменными-членами. Щелкнув какой-нибудь компонент, вы тут же увидите соответствующий исходный код. Однако в отличие от средства просмотра режим Class View не позволяет отображать информацию об иерархии.

Solution Explorer

В Solution Explorer отображается структура всего проекта. Приложение в Visual Studio .NET может состоять из многих элементов, в том числе из многих проектов. Solution Explorer позволяет управлять всеми элементами решения.

Окно Solution Explorer содержит древовидное представление элементов проекта, которые можно открывать по отдельности для модификации или выполнения задач по управлению. В дереве отображаются логические отношения решения и проектов, а также элементов решения. Чтобы связать файлы с решением, но не с одним из его проектов, достаточно присоединить его прямо к решению.

Object Browser

Visual C++ .NET Object Browser позволяет изучать (и редактировать) приложение с точки зрения классов и функций, а не отдельных физических файлов. В этом смысле он напоминает инструменты-инспекторы, имеющиеся во многих объектно-ориентированных библиотеках, в частности Smalltalk.

Чтобы открыть окно Object Browser, в подменю Other Windows главного меню View выберите команду Object Browser. В Object Browser несколько режимов просмотра.

- **Definitions and references** Можно выбрать любую функцию, переменную, тип, макрос или класс и посмотреть, где она определена и используется в проекте.
- **Sorting** Сортировка объектов и членов по алфавиту, по типу или виду доступа.
- **Derived Classes and Members/Base Classes and Members** Графические схемы иерархии классов. У выбранного класса можно посмотреть производные и базовые классы с их членами. Уровень иерархии выбирается мышью.

Стандартное окно утилиты Object Browser представлено в главе 3.

Примечание Если изменить порядок строк исходного кода, Visual C++ .NET обновит базу данных Object Browser при следующей сборке проекта. Это займет некоторое время.

UML-инструменты

Теперь в Visual C++ .NET появились инструменты моделирования на языке UML (Unified Modeling Language), представляющем собой набор соглашений о созда-

нии диаграмм и описаний программы, описывающих систему. Среди поддерживаемых типов UML-схем диаграммы классов, объектов, действий и состояний. Во многих организациях UML применяется как стандартное средство документирования систем.

В Visual C++ .NET есть команда в меню Project для обратного преобразования проекта в UML-диаграмму. Для обратного преобразования проекта Visual C++ .NET в набор UML-диаграмм сначала надо собрать информацию о проекте для Object Browser, а затем последовательно выбрать команды Visio UML и Reverse Engineer в меню Project. Visual C++ .NET создаст UML-пакет (набор диаграмм) проекта, откроет Visio и отобразит в нем пакет. (UML-диаграммы создаются в формате Visio.)

Примечание Для просмотра раздела интерактивной справочной системы, посвященного UML-решениям Visio, должно быть открыто приложение Visio. В конце процедуры установки Visual Studio .NET Enterprise Architect мастер предлагает установить Visio.

Интерактивная справочная система

В версии 6 среды Visual C++ справочная система перенесена в формат HTML и выделена в отдельное приложение — MSDN Library Viewer. Темы размещаются в отдельных HTML-документах, а сами документы компилируются в проиндексированные файлы. В MSDN Library Viewer используется код из Microsoft Internet Explorer 4.0, и поэтому справочная система работает, как хорошо знакомый вам Web-браузер. Справочная система обеспечивает доступ к справочным файлам, находящимся как на компакт-диске Visual C++ .NET (выбор по умолчанию), так и на жестком диске, а также к HTML-файлам в Интернете. Получить справку можно по-разному.

- **По книге** Команда Contents меню Help среды разработки открывает окно Contents, отображающее информацию о документации Visual Studio .NET и библиотеке MSDN. Документация по Visual Studio .NET, .NET Framework SDK и Platform SDK представлена иерархически по книгам и главам. Содержание определяется выбранными фильтрами.
- **По теме** Команда Index меню Help среды разработки открывает окно Index. Чтобы получить список относящихся к тому или иному ключевому слову тем и статей, достаточно ввести его в поле и выполнить поиск. Содержание определяется выбранными фильтрами.
- **По слову** Команда Search меню Help среды разработки открывает окно Search, используемое для полнотекстового поиска документов, в которых встречаются определенные слова. Содержание определяется выбранными фильтрами.
- **Динамическая справка** Позволяет получать от Visual Studio .NET ссылки на информацию о конкретной области, в которой вы работаете, или задачи, которую пытаетесь выполнить в IDE-среде.
- **Контекстная справка по нажатию клавиши F1** Лучший друг программиста. Поместите указатель на имя функции, макроса или класса, нажмите клавишу F1, и справочная система примется за работу. Если имя встречается в нескольких местах (скажем, в справочных файлах MFC и Win32), откроется окно

Index со списком соответствующих тематических разделов, из которых можно выбрать нужный.

Независимо от способа получения любой фрагмент справки можно скопировать в буфер обмена и перенести, например, в программу.

Диагностические утилиты

В Visual C++ .NET есть ряд полезных диагностических инструментов. SPY++ показывает дерево процессов, потоков и окон, существующих в системе. Он позволяет также просматривать сообщения и исследовать окна исполняемых приложений. В Visual C++ .NET имеется и богатый набор ActiveX-утилит, программа для тестирования элементов управления ActiveX и другие инструменты.

Библиотека MFC версии 7

Библиотека MFC версии 7 определяет каркас приложений, с которым вам надо познакомиться поближе. В главе 2 вы увидите настоящий код и узнаете ряд важных положений.

Библиотека ATL версии 7

Библиотека ATL отделена от MFC и применяется для создания элементов управления ActiveX. Вообще писать элементы управления ActiveX можно как на MFC, так и на ATL, но ATL-элементы гораздо меньше по объему кода и быстрее загружаются через Интернет. В главах 27 и 28 вы познакомитесь с ATL и методиками создания элементов управления ActiveX средствами ATL.

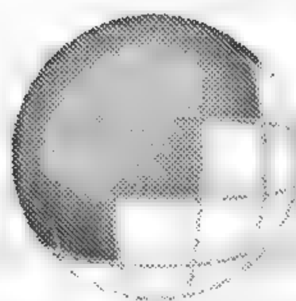
Поддержка .NET

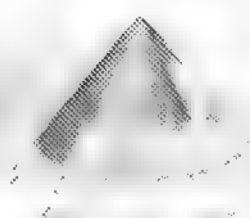
Visual Studio .NET полностью поддерживает каркас .NET Framework. Хотя DLL-библиотеки, C++, библиотеки MFC, COM и ATL можно применять совместно для разработки Windows-приложений, у созданной таким образом системы появляется ряд неприятных свойств. Иногда кажется, что связи некоторых частей слишком искусственны. Одна из главных задач .NET — унификация модели программирования, т. е. обеспечение большего единства платформы Windows. CLR-среда предполагает единый набор типов в синтаксисе всех программ. ASP.NET также работает под управлением CLR, так что разработка Web-приложений тоже унифицируется.

Кроме того, управляемый код есть не только в Visual Basic .NET — Microsoft расширила C++, добавив поддержку управляемого кода — *управляемый C++* (Managed Extensions), позволяющий создавать код, исполняемый CLR-средой. Очень много особенностей кода на C++ сохранено, и предполагается, что Managed Extensions облегчит переход в среду .NET. Подробно о .NET и роли Visual C++ .NET в создании приложений .NET мы расскажем во второй части.

ГЛАВА

2





Каркас приложений Microsoft Foundation Class Library

В этой главе вы познакомитесь с каркасом приложений Microsoft Foundation Class Library версии 7.0 (библиотеки MFC). В следующих главах вам встретятся хотя и урезанные, но вполне работоспособные Windows-программы на базе MFC, которые помогут уяснить, как же программируют с применением каркаса приложений. Мы постарались свести к минимуму объем теоретических сведений, однако включили дополнительные разделы, посвященные сопоставлению сообщений, а также документам и видам, — они помогут осмыслить примеры из следующих глав.

Назначение каркаса приложений

Чтобы создать приложение для Windows, надо выбрать среду разработки. И если вы уже отвергли варианты, не связанные с языком С (скажем, Microsoft Visual Basic или Borland Delphi), у вас все же остается несколько путей:

- программировать на С, применяя Win32 API;
- написать на С++ свою библиотеку классов, использующую Win32;
- задействовать каркас приложений на базе MFC;
- выбрать другой каркас приложений, например Object Windows Library (OWL) фирмы Borland (Inprise).

Примечание О программировании с помощью .NET Windows Forms см. часть 6.

Если вы начинаете с нуля, вам потребуется очень многому учиться. Вы уже умеете программировать Win32? Хорошо, но все равно придется осваивать библиотеку MFC. Дело в том, что MFC очень быстро стала доминирующей библиотекой клас-

сов для Windows. Даже если вы знакомы с MFC, не помешает еще раз вспомнить основные особенности этого программного продукта.

Библиотека MFC — интерфейс программирования на C++, расположенный поверх API Windows. Язык C++ сегодня является стандартом для разработки серьезных приложений и пользуется мощной поддержкой самых разных компаний. Чтобы создавать максимально производительные приложения, нужно программировать максимально близко к Windows API. C++ и MFC предоставляют такую возможность, избавляя от написания методов *WndProc* вручную.

«Каркасные» приложения имеют стандартную структуру. Приступая к крупному проекту, нужно выработать структуру кода. Но эта структура у каждого программиста своя, и новому участнику команды трудно приспособиться к ней. Каркас приложений на базе MFC предлагает свою структуру, отработанную на многих проектах и опробованную в разных программных средах. Создав Windows-программу на базе библиотеки MFC, можно спокойно удалиться хоть на Карибы — оставшиеся дома без проблем смогут поддерживать и совершенствовать вашу программу.

Не подумайте только, что структура библиотеки MFC сковывает программу, лишая ее гибкости. При работе с MFC можно в любой момент вызвать любую Win32-функцию, т. е. в полной мере задействовать возможности и преимущества Windows.

Достоинства «каркасных» приложений — компактность и высокая скорость работы. Во времена 16-разрядного программирования можно было создать исполняемый файл Windows-программы размером меньше 20 кб. Сегодня Windows-программы гораздо больше, и одна из причин в том, что 32-разрядный код объемнее. Даже используя бóльшую модель памяти, Win16-программы оперировали с 16-разрядными адресами стековых и многих глобальных переменных. А Win32-программы всегда используют 32-разрядные адреса и часто работают с 32-разрядными целыми значениями — они эффективнее 16-разрядных. Много памяти расходует и новый код обработки исключений в C++.

Вспомните: разве в тех 20-килобайтовых программах были *стыкуемые панели инструментов* (docking toolbars), разделяемые окна, режим предварительного просмотра перед печатью, поддержка контейнеров с элементами управления — словом, все те возможности, которых пользователи ждут от современных программ? MFC-программы потому и объемнее, что делают больше и выглядят лучше. К счастью, теперь можно создавать программы, динамически связываемые с MFC-кодом (и C-кодом периода исполнения), так что их размер опять уменьшается — со 192 кб до примерно тех же 20 кб. Разумеется, такой программе понадобится масса DLL-модулей, но в наши дни от этого никуда не деться.

Что до скорости работы программ, то вы имеете дело с машинным кодом, который сгенерирован оптимизирующим компилятором. Программы исполняются быстро, но при их запуске можно заметить задержку — в этот момент загружаются вспомогательные DLL-модули.

Средства Visual C++ .NET уменьшают необходимость в написании рутинного кода. Редакторы ресурсов, MFC Application Wizard и мастера средства Class View значительно ускоряют написание кода, характерного для конкретного приложения. Например, редактор ресурсов создает заголовочный файл с уже определенными значениями для *#define*-констант. MFC Application Wizard генерирует

«скелет» всего приложения, после чего в окне свойств можно добавлять обработчики сообщений и сопоставлять им сообщения.

У библиотеки MFC огромное количество разнообразных возможностей. Классы библиотеки MFC версии 1.0, поставлявшиеся с Microsoft C/C++ версии 7.0, поддерживали следующие возможности:

- интерфейс C++ с Windows API;
- классы общего назначения (не относящиеся непосредственно к Windows), в том числе:
  - наборы классов для списков, массивов и карт (maps);
  - удобный и эффективный строковый класс;
  - классы для работы со временем, временными интервалами и датами;
  - классы для независимого от ОС доступа к файлам;
  - поддержка систематизации сохранения объектов на диске и их считывания с диска;
- иерархия классов от общего корневого объекта;
- поддержка приложений с многодокументным интерфейсом (Multiple Document Interface, MDI);
- поддержка OLE версии 1.0.

Вторая версия (включенная в Visual C++ 1.0) приняла эстафету у первой версии: введены поддержка функций пользовательского интерфейса современных Windows-приложений, а также каркасная архитектура приложений. В MFC 2.0 появились:

- полная поддержка команд Open, Save, Save As в меню File и списка последних открывавшихся файлов;
- предварительный просмотр перед печатью и поддержка принтера;
- поддержка окон с прокруткой и разбиением;
- поддержка панелей инструментов и строк состояния;
- доступ к элементам управления Visual Basic;
- поддержка контекстно-зависимой справки;
- поддержка автоматической обработки данных, вводимых в диалоговом окне;
- улучшенный интерфейс с OLE 1.0;
- поддержка DLL.

Классы версии 2.5 (в Visual C++ версии 1.5) ввели в библиотеку MFC:

- поддержку механизма ODBC, что позволило приложениям оперировать информацией, хранящейся в таких базах данных, как Microsoft Access, FoxPro и Microsoft SQL Server;
- интерфейс с OLE 2.01, включая поддержку редактирования «по месту», связывание, перемещения объектов мышью (drag and drop), а также OLE Automation.

Visual C++ 2.0 была первой 32-разрядной версией продукта и поддерживала Windows NT 3.5. В нее входила MFC версии 3.0 с такими новыми возможностями:

- поддержка диалоговых окон с вкладками (входила и в версию 1.51 Visual C++, поставляемую на том же компакт-диске);
- стыкуемые панели элементов управления, реализованные в MFC;

- поддержка окон с тонкими рамками (thin-frame windows);
- отдельный CDK (Control Development Kit) для построения 16- и 32-разрядных элементов управления на базе OLE, хотя поддержка OLE-контейнеров элементов управления еще не предусматривалась.

В последующей версии — Visual C++ 2.1 с MFC 3.1 — добавились:

- поддержка новых стандартных элементов управления из бета-версии Windows 95;
- драйвер ODBC Level 2, совместимый с ядром баз данных Access Jet;
- классы Winsock для обмена данными по протоколу TCP/IP.

Далее Microsoft решила «перескочить» через третью версию Visual C++ и приступила прямо к четвертой — чтобы синхронизировать номера версий Visual C++ и MFC. В библиотеке MFC 4.0 появились:

- новые классы объектов доступа к данным на базе OLE (Data Access Objects, DAO) для работы с Jet;
- стыкуемые панели элементов управления Microsoft Windows 95 вместо панелей элементов управления MFC;
- полная поддержка стандартных элементов управления из финальной версии Windows 95 с новыми классами *древовидный список* (tree view) и *поле ввода с форматированием* (rich-edit view);
- новые классы для синхронизации потоков;
- поддержка OLE-контейнеров элементов управления.

Важным этапом стала версия Visual C++ 4.2 с MFC 4.2 и такими возможностями:

- классы WinInet;
- классы серверов ActiveX-документов;
- классы синхронных и асинхронных моникеров ActiveX;
- усовершенствованные MFC-классы элементов управления ActiveX, с такими возможностями, как активизация без образования окна, оптимизированный код рисования и т. д.;
- улучшенная поддержка MFC ODBC, включая массированную загрузку наборов данных и пересылку данных без привязки.

В Visual C++ 5.0 с MFC 4.21, в которой было исправлено несколько ошибок версии 4.2, ввели еще несколько заслуживающих внимания возможностей:

- обновленная среда разработки, Developer Studio 97; ее особенности — основанная на HTML справочная система и интеграция с другими языками, в том числе с Java;
- Active Template Library (ATL) для эффективной разработки элементов управления ActiveX для Интернета;
- поддержка в языке C++ при помощи директивы *#import* программирования COM-клиентов на основе библиотек типов (подробнее — в главе 25).

Visual C++ 6.0 содержит MFC 6.0 (заметьте: номера версий вновь синхронизированы). Многие из возможностей MFC 6.0 обеспечивали поддержку передовой на то время концепции Microsoft Active Platform:

- классы MFC, инкапсулирующие новые стандартные элементы управления Windows, которые являются составной частью в Internet Explorer 4.0;
- поддержка Dynamic HTML, позволяющая MFC-программисту разрабатывать приложения, способные динамически изменять и создавать HTML-страницы;
- технология Active Document Containment, позволяющая приложениям, основанным на MFC, включать активные документы (Active Documents);
- поддержка шаблонов поставщиков и потребителей OLE DB и привязки данных ADO, что очень удобно для разработчиков программ доступа к базам данных, использующих MFC или ATL.

Последняя версия, Visual C++ .NET, содержит библиотеку MFC 7.0, в которой обеспечена поддержка возможностей Интернет-программирования (и на новой платформе Microsoft .NET), а также улучшены процедуры программирования для Windows. Среди новых возможностей:

- усовершенствованная поддержка справочной системы на основе HTML в MFC-приложениях;
- поддержка безоконных элементов управления;
- диалоговые окна и компоненты-редакторы в DHTML;
- классы обработки аргументов в HTML;
- диалоговое окно печати в Windows 2000;
- более жесткий контроль типов в сообщениях;
- поддержка дат, следующих за 2038 годом.

Путь, который вам предстоит

Все это здорово, но вы, вероятно, подумали: «За так ничего не получишь». Верно. Чтобы эффективно пользоваться каркасом приложений, надо досконально изучить его, а это время. И если вам придется осваивать и C++, и Windows, и библиотеку MFC (без OLE), то сделать что-то полезное вы сможете не раньше, чем через полгода. Что интересно — на изучение одного только Win32 API уходит примерно столько же времени.

Как же так, спросите вы, если библиотека MFC предлагает столько возможностей? Ну, прежде всего вам не избежать многого из того, чему вынуждены учиться Win32-программисты на C. Исходя из опыта, можем сказать, что объектно-ориентированный каркас приложений упрощает обучение программированию для Windows, если, конечно, вы разбираетесь в объектно-ориентированном программировании.

Безусловно, библиотека MFC не сделает программирование для Windows достоянием масс. Здесь надо сказать, что «стоимость» Windows-программистов на рынке труда выше, чем других, и вряд ли эта ситуация изменится в ближайшее время. Широкий спектр возможностей MFC вкупе с мощью каркаса приложений служит гарантией сохранения высокого спроса на программистов, умеющих работать с MFC.

Каркас приложений

Одно из определений термина *каркас приложений* (application framework) таково: «интегрированный набор объектно-ориентированных программных компонентов, обеспечивающих все, что нужно для работы программы общего назначения». Туманно, не так ли? Если вы действительно хотите знать, что такое каркас приложений, вам придется дочитать книгу до конца. Первым вам на помощь придет пример, с которым мы познакомимся чуть позже.

Каркас приложений и библиотека классов

Одна из причин популярности C++ в том, что этот язык можно «расширять» библиотеками классов. Одни библиотеки поставляются с компиляторами C++, другие продают независимые фирмы, а какие-то создают сами программисты, так сказать, для внутреннего потребления. Библиотека классов — это набор взаимосвязанных классов C++, которые можно задействовать в приложении. Например, математическая библиотека классов выполняет наиболее распространенные математические операции, а библиотека коммуникационных классов поддерживает обмен данными по последовательному каналу. Иногда вы конструируете объекты из предлагаемых классов, иногда создаете из них собственные классы — все зависит от конструкции конкретной библиотеки.

Каркас приложений — это надмножество библиотеки классов. Обычная библиотека представляет собой изолированный набор классов, предназначенных для применения в любой программе, а каркас приложений определяет структуру самой программы. Концепцию каркаса приложений изобрела не Microsoft — она сначала появилась в академических кругах, а первым коммерческим воплощением стала MacApp для Apple Macintosh. С появлением MFC 2.0 реализацией подобных продуктов занялись и другие фирмы, в том числе Borland (Inprise).

Пример приложения на базе каркаса приложений

Хватит общих рассуждений. Пора взглянуть на какой-нибудь код — не псевдо, а настоящий, который действительно можно скомпилировать и выполнить с библиотекой MFC. Угадайте, какой? Ну конечно, это старая добрая программа «Hello, world!», правда, с некоторыми дополнениями. (Если вам доводилось работать с первой версией библиотеки MFC, то этот код вам знаком, кроме базового класса окна-рамки.) Кстати, это почти минимум кода, необходимого для Windows-приложения с применением библиотеки MFC. Сравните его с аналогичным чисто Win32-приложением из книги Ч. Петцольда! Пока не пытайтесь разобраться во всех деталях. Не набирайте и не тестируйте его, потому что пример Ex21b на компакт-диске — практически полная копия. Потерпите до следующей главы — там-то вы и начнете работать с «реальным» каркасом приложений.

Примечание Имена классов библиотеки MFC принято начинать с буквы C.

Ниже приведен исходный код для заголовочного файла и *файла реализации* (implementation file) нашего приложения MyApp. Два класса — *CMyApp* и *CMyFrame* — наследуют базовым классам библиотеки MFC. Сначала взгляните на заголовочный файл MyApp.H для приложения MyApp:

```
// Класс приложения
class CMyApp : public CWinApp
{
public:
   virtual BOOL InitInstance();
};

// Класс окна-рамки
class CMyFrame : public CFrameWnd
{
public:
   CMyFrame();
protected:
   // "afx_msg" означает, что следующие две функции являются
   // частью системы маршрутизации сообщений библиотеки MFC
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   afx_msg void OnPaint();
   DECLARE_MESSAGE_MAP()
};
```

А вот файл реализации MyApp.CPP для приложения MyApp:

```
#include <afxwin.h>    // Заголовочный файл библиотеки MFC,
                       // в котором объявлены базовые классы
#include "myapp.h"

CMyApp theApp; // Единственный объект CMyApp

BOOL CMyApp::InitInstance()
{
   m_pMainWnd = new CMyFrame();
   m_pMainWnd->ShowWindow(m_nCmdShow);

   m_pMainWnd->UpdateWindow();
   return TRUE;
}

BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)
   ON_WM_LBUTTONDOWN()
   ON_WM_PAINT()
END_MESSAGE_MAP()

CMyFrame::CMyFrame()
{
   Create(NULL, "MYAPP Application");
}

void CMyFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
   TRACE("Entering CMyFrame::OnLButtonDown - %lx, %d, %d\n",
       (long) nFlags, point.x, point.y);
}
```

```
void CMyFrame::OnPaint()
{
   CPaintDC dc(this);
   dc.TextOut(0, 0, "Hello, world!");
}
```

Теперь обсудим некоторые элементы программы.

- **Функция *WinMain*.** Windows требует наличия этой функции в любой программе. Здесь вы ее не видите потому, что она скрыта внутри каркаса приложения.
- **Класс *CMyApp*.** Объект класса *CMyApp* представляет программу. В программе определяется единственный глобальный объект класса *CMyApp* — *theApp*. Базовый класс *CWinApp* определяет основные характеристики поведения объекта *theApp*.
- **Запуск приложения.** При запуске приложения Windows вызывает встроенную в каркас приложений функцию *WinMain*, а та ищет глобально сконструированный объект класса, производного от *CWinApp*. Не забудьте, что в программах на C++ глобальные объекты конструируются *перед* исполнением основной части программы.
- **Функция-член *CMyApp::InitInstance*.** Найдя объект-приложение, *WinMain* вызывает виртуальную функцию-член *InitInstance*, которая делает вызовы, необходимые для создания и отображения на экране *основного окна-рамки* (main frame window) приложения. Вы должны переопределить *InitInstance* в своем производном классе «приложение», так как базовый класс *CWinApp* не имеет представления о том, какого типа основное окно-рамку вы хотите создать.
- **Функция-член *CWinApp::Run*.** Функция *Run* скрыта в базовом классе, но она распределяет сообщения программы между ее окнами, обеспечивая тем самым работу этой программы. *WinMain* вызывает *Run* после вызова *InitInstance*.
- **Класс *CMyFrame*.** Объект класса *CMyFrame* представляет основное окно программы. Когда конструктор вызывает функцию-член *Create* базового класса *CFrameWnd*, Windows создает настоящую оконную структуру, а каркас приложения связывает ее с объектом C++. Для отображения окна на экране вызываются функции *ShowWindow* и *UpdateWindow* (это также функции-члены базового класса).
- **Функция *CMyFrame::OnLButtonDown*.** Включена для демонстрации возможностей обработки сообщений в библиотеке MFC. Мы объявляем о сопоставлении события «нажатие левой кнопки мыши» с функцией-членом класса *CMyFrame*. Подробнее сопоставление сообщений в библиотеке MFC мы обсудим в главе 5, а пока просто примем к сведению, что эта функция вызывается при нажатии левой кнопки мыши. Функция использует макрос *TRACE* из библиотеки MFC, чтобы вывести сообщение в отладочное окно.
- **Функция *CMyFrame::OnPaint*.** Каркас приложений вызывает эту функцию-член класса *CMyFrame* всякий раз, когда надо перерисовать окно: в начале работы программы, при изменении размеров окна и при обновлении всего окна или его части. Оператор *CPaintDC* относится к «классическому» GDI-интерфейсу и поясняется в следующих главах. Ну, а функция *TextOut* выводит на экран «Hello, world!». (С GDI+ мы познакомимся поближе при обсуждении .NET)

- **Завершение приложения.** Пользователь завершает приложение, закрывая основное окно программы. Тем самым он инициирует последовательность событий, заканчивающихся уничтожением объекта класса *CMyFrame*, выходом из *Run*, выходом из *WinMain* и уничтожением объекта класса *CMyApp*.

Еще раз взгляните на пример и попробуйте теперь представить картину целиком. Очевидно, что большая часть возможностей программы сосредоточена в базовых классах библиотеки MFC: *CWinApp* и *CFrameWnd*. Составляя MYAPP, мы следовали нескольким простым правилам структуризации и поместили ключевые функции в производные классы. Как видите, C++ позволяет «заимствовать» большие объемы кода, не копируя его. Считайте это партнерскими отношениями между нами и каркасом приложений. Последний создает структуру программы, а мы — код, наполняющий эту структуру конкретным смыслом.

Теперь вы, наверное, начинаете понимать, почему каркас приложения — нечто большее, чем библиотека классов. Он определяет не только структуру программы — его роль значительнее, чем у базовых классов C++. Вы уже видели в действии скрытую функцию *WinMain*, а прочие компоненты каркаса поддерживают обработку сообщений, динамически подключаемые библиотеки и многое другое.

Сопоставление сообщений в библиотеке MFC

Вспомним функцию-член *OnLButtonDown* из предыдущего примера. Можно подумать, что *OnLButtonDown* — идеальный кандидат на роль виртуальной функции. При таком подходе базовый класс окна определил бы виртуальные функции для сообщений о событиях, связанных с мышью, и других стандартных сообщений, а производные классы окна могли бы при необходимости переопределять эти функции. Некоторые библиотеки классов для Windows именно так и устроены.

Но в каркасе приложений библиотеки MFC не используются виртуальные функции для сообщений Windows. Вместо этого с помощью макросов определенные сообщения сопоставляются (map) функциям-членам производных классов. Чем же объясняется отказ от виртуальных функций? Допустим, виртуальные функции для сообщений в MFC все-таки применяются. Класс *CWnd* должен был бы объявлять виртуальные функции для не менее сотни сообщений. Для каждого производного класса, задействованного в программе, C++ требует наличия *таблицы диспетчеризации виртуальных функций* (vtable) (виртуальная таблица). Для каждой виртуальной функции в этой таблице потребуется 4-байтовая запись независимо от того, переопределена ли функция в производном классе. Так что для каждого типа окна или элемента управления приложению для поддержки виртуальных обработчиков сообщений понадобится таблица размером более 400 байт.

А как насчет обработчиков сообщений о выборе команд меню и щелчках кнопок мыши? Их не определишь как виртуальные функции в базовом классе окна, так как в каждом приложении свой набор команд меню и кнопок. Система карт сообщений, принятая в библиотеке MFC, позволяет обойтись без длинных виртуальных таблиц и сглаживает различия между обычными Windows-сообщениями и командными сообщениями, характерными для конкретных программ. Поэтому отдельные классы, отличные от оконных (классы «документ» и «приложение»), могут обрабатывать и командные сообщения. Для сопоставления (или увязки)

сообщений Windows с функциями-членами C++ в MFC применяются макросы. Расширения языка C++ не нужны.

MFC-обработчик сообщения требует наличия прототипа функции, тела функции и соответствующей записи в карте сообщений. Вставлять обработчики сообщений в классы можно в окне Properties. Вы выбираете идентификатор Windows-сообщения из списка, а мастер генерирует код с нужными параметрами функций и возвращаемыми значениями.

Документы и их представление

В нашем примере мы использовали объект-приложение и объект «окно-рамка». Но большинство реальных приложений на базе библиотеки MFC гораздо сложнее. Обычно они, помимо упомянутых, содержат еще два класса: «документ» (document) и «вид», или «представление» (view). Архитектура «документ-вид» — стержень каркаса приложений; она напоминает классы Model/View/Controller, принятые в среде Smalltalk.

Проще говоря, архитектура «документ-вид» отделяет данные от их представления пользователю. Очевидное преимущество этого подхода — возможность представить одни и те же данные по-разному. Пусть на диске хранится документ со сводкой котировок за месяц, а данные представляются в виде таблицы и графика. Мы изменяем значения в окне табличного представления, и содержимое окна графического представления тоже изменяется, так как оба окна отображают одну и ту же информацию (но представленную в разных видах).

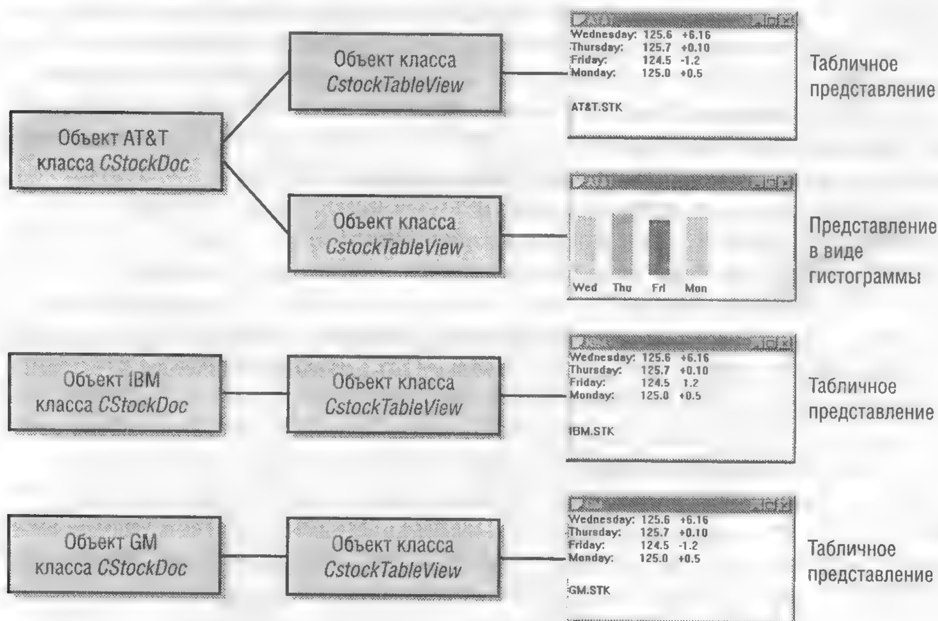


Рис. 2-1. *Взаимосвязь документов и их представления*

В библиотеке MFC документам и их видам сопоставляются *экземпляры* (instances) классов C++. Так, на рис. 2-1 показаны три объекта класса *CStockDoc*, соответ-

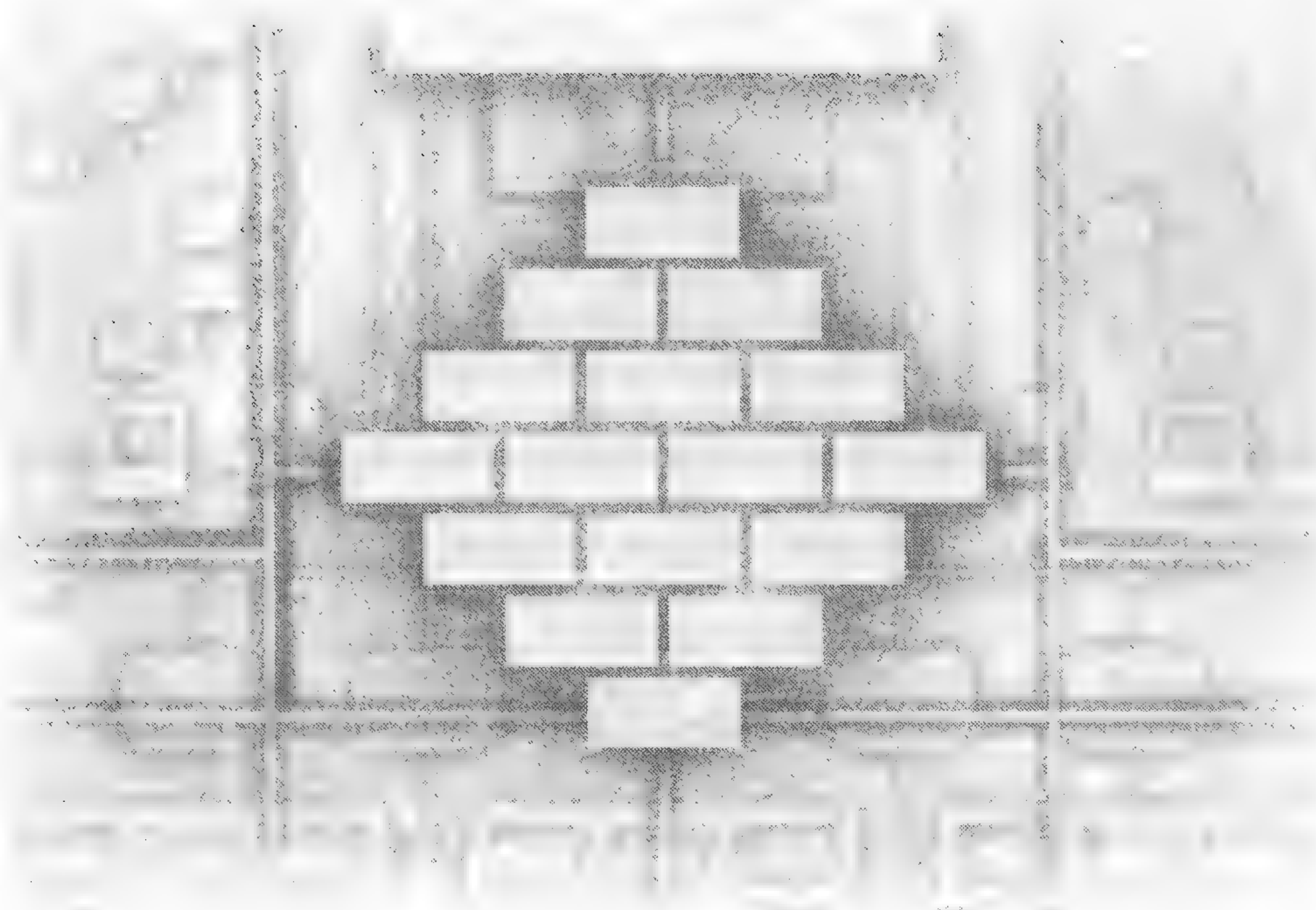
ствующие трем фирмам: *AT&T*, *IBM* и *GM*. Все три документа связаны с табличным представлением, а один — еще и с графическим (гистограммой). Как видите, здесь четыре объекта «вид»: три объекта класса *CstockTableView* и один — класса *CStockChartView*.

Код базового класса «документ» взаимодействует с командами Open и Save меню File; само чтение и запись данных объекта-документа реализовано в производных классах «документ». (Каркас приложений берет на себя большую часть работы по выводу на экран диалоговых окон File Open и File Save, а также по открытию, закрытию, чтению и записи файлов.) Базовому классу «вид» сопоставлено окно, содержащееся внутри окна-рамки; производный класс «вид» взаимодействует со своим, сопоставленным ему классом «документ» и отвечает за операции ввода/вывода информации на экран и принтер. Производный класс «вид» и его базовые классы обрабатывают сообщения Windows. Библиотека MFC «дирижирует» взаимодействием документов, их представлениями, окнами-рамками и объектом-приложением в основном посредством виртуальных функций.

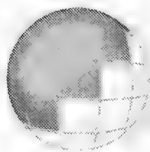
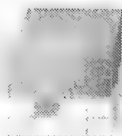
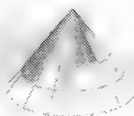
Не подумайте, что объект-документ надо обязательно связывать с дисковым файлом, целиком считываемым в память. Если, например, «документ» — на самом деле база данных, можно переопределить нужные функции-члены класса «документ», и тогда по команде Open меню File отобразится список баз данных, а не файлов.

ЧАСТЬ II

ОСНОВЫ MFC



ГЛАВА 3

Знакомство с мастером создания MFC-приложения

В главе 2 мы в общих чертах познакомились с архитектурой «документ-вид» библиотеки MFC. В данной главе вы увидите, как создать приложение на базе функций этой библиотеки, не вдаваясь в сложности иерархии классов и взаимосвязей объектов. Вы поработаете только с одним компонентом программы — классом «вид» (view class), тесно связанным с объектом «окно». Такие компоненты, как класс «приложение» (application class), «окно-рамка» (frame window) и «документ», можно пока игнорировать. Ваша программа, конечно, не сможет сохранять свои данные на диске и не будет поддерживать множественное представление информации — о том, как это сделать, и о многом другом вы прочтете в части 3.

Поскольку в Windows-приложениях важную роль играют ресурсы, вы будете использовать Resource View для визуального просмотра ресурсов создаваемой программы. Кроме того, вы получите несколько советов по настройке среды Windows для обеспечения максимальной скорости сборки программы, а также по оптимизации вывода отладочной информации.

Примечание Чтобы скомпилировать и запустить примеры программ этой и следующих глав, надо установить на компьютере Microsoft Windows NT 4.0, Windows 2000 или Windows XP, а также все компоненты Microsoft Visual C++ .NET. Проверьте правильность настройки каталогов с исполняемыми файлами, библиотеками и включаемыми файлами среды Visual C++ .NET. (Изменить пути к каталогам позволяет команда Options из меню Tools.) Если возникнут трудности, обратитесь к документации Visual C++ .NET и файлам README.

Что такое «вид»

С точки зрения пользователя, *вид* (view) — это обычное окно, т. е. вы можете изменять его размеры, перемещать и закрывать, как любые окна в приложениях Windows. А с точки зрения программиста, это объект класса C++, производного от класса *CView* библиотеки MFC. Как и для любого объекта C++, поведение объекта «вид» определяется функциями-членами (и переменными-членами) соответствующего класса, включая и специфичные для приложения функции производного класса, и стандартные функции, унаследованные от базовых классов.

Visual C++ .NET позволяет создавать достаточно интересные Windows-приложения, просто добавляя код в производный класс «вид», сгенерированный мастером MFC Application Wizard. Во время работы вашей программы каркас приложения MFC создает объект производного класса «вид» и отображает окно, тесно связанное с этим объектом C++. Как принято в C++, код класса размещается в двух исходных модулях: заголовочном файле (H) и файле реализации (CPP).

Типы MFC-приложений

Библиотека MFC поддерживает приложения трех типов: с *однодокументным* (Single Document Interface, SDI) и *многодокументным* (Multiple Document Interface, MDI) *интерфейсами*, а также с *интерфейсом на основе многих окон верхнего уровня* (Multiple Top-Level Windows Interface, MTI). С точки зрения пользователя, SDI — это приложение, состоящее из единственного окна. Работая с «документами» в дисковых файлах, в каждый момент времени оно способно загрузить только один документ. Пример SDI-приложения — Notepad (Блокнот). В MDI-приложении есть несколько *дочерних окон* (child windows), в каждое из которых загружается свой документ. Прекрасный пример такого приложения — версии Microsoft Word, предшествующие Microsoft Word 2000.

В мастере MFC Application Wizard по умолчанию создается MDI-приложение. В начальных примерах этой книги мы будем создавать SDI-приложения, так как в них меньше классов и требуется учитывать меньше параметров. Каждый раз вам придется позаботиться о выборе в качестве типа приложения Single Document в первом диалоговом окне MFC Application Wizard. С главы 18 мы будем создавать MDI-приложения. Архитектура каркаса приложений MFC позволяет легко преобразовывать большинство SDI-примеров в MDI-приложения.

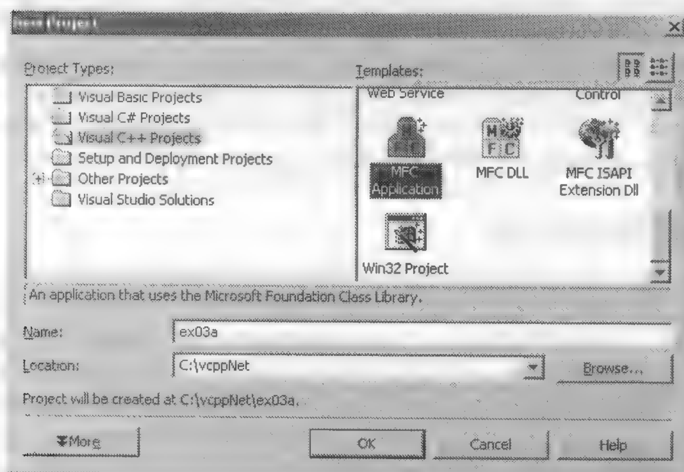
Пользовательские интерфейсы MFC-приложений

Кроме SDI-, MDI- и MTI-стилей пользовательского интерфейса, библиотека MFC позволяет создавать приложения со стандартным пользовательским интерфейсом или интерфейсом в стиле Windows Explorer. Примеры приложений с «классическим» интерфейсом — Microsoft Word и Microsoft Paintbrush. Интерфейс в стиле Windows Explorer состоит из двух панелей: левая содержит древообразный вид с развертываемыми узлами, правая — представление в виде списка. Естественно, что Windows Explorer — наглядный пример подобного интерфейса.

Пример Ex03a: «пустое» приложение

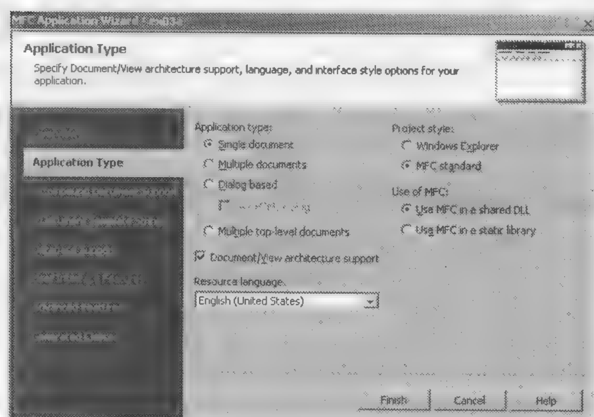
MFC Application Wizard генерирует код работающего MFC-приложения, которое просто отображает на экране пустое окно с меню. Позже вы добавите код, «рисующий» внутри окна, а пока просто создадите простейшее приложение.

1. **Сгенерируйте код SDI-приложения средствами MFC Application Wizard.** В подменю New меню File выберите команду Project, в открывшемся диалоговом окне выберите узел Visual C++ Projects, а в списке шаблонов — MFC Application:



В поле Location введите путь **C:\vcppNet**, а в поле Project Name — **Ex03a** и щелкните OK. Ссылки левой панели позволяют перемещаться по страницам мастера MFC Application Wizard и настраивать параметры будущего приложения.

На странице Application Type выберите вариант Single document и оставьте без изменений остальные свойства приложения:



Заметьте: на странице Generated Classes имена классов и файлов соответствуют названию приложения — Ex03a. На данном этапе эти имена можно изменить. Щелкните Finish. Мастер создаст подкаталог приложения (ex03a в

каталоге \vcppNet), а в нем ряд файлов. По окончании работы мастера посмотрите на содержимое каталога приложения (табл. 3-1).

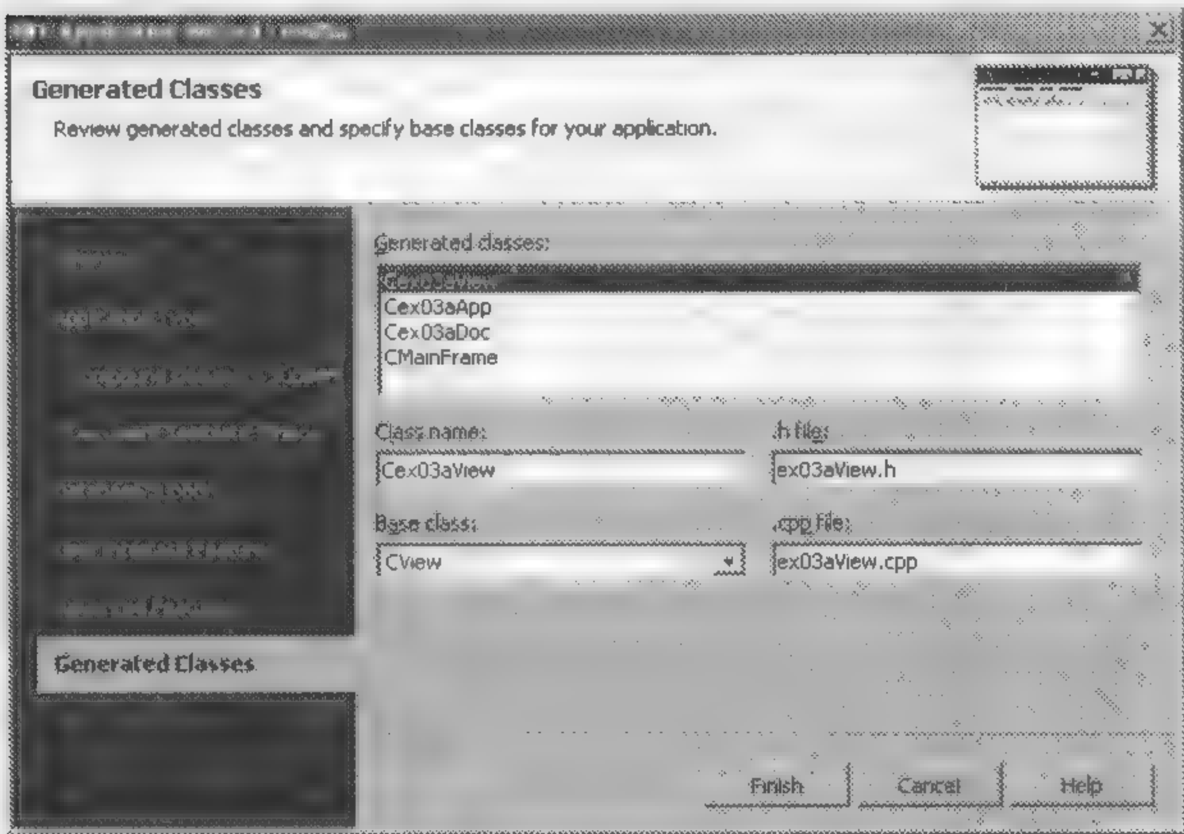


Табл. 3-1. Важные файлы в подпапке приложения

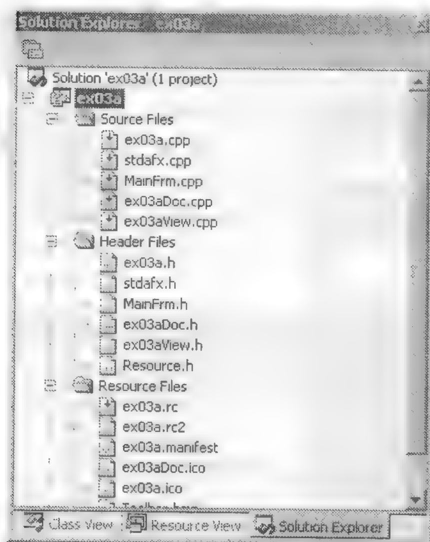
Файл	Описание
Ex03a.vsproj	Файл проекта, применяемый Visual C++ .NET для сборки приложения
Ex03a.sln	Файл решения с единственной записью о проекте Ex03a.vsproj
Ex03a.rc	Текстовый файл описаний ресурсов
Ex03aView.cpp	Файл реализации класса «вид» с функциями-членами класса *CEx03aView*
Ex03aView.h	Заголовочный файл класса «вид», содержащий объявление класса *CEx03aView*
ReadMe.txt	Текстовый файл с описанием назначения сгенерированных файлов
Resource.h	Заголовочный файл, содержащий определения констант *#define*

Откройте файлы ex03aView.cpp и ex03aView.h и взгляните на исходный код. В совокупности эти файлы определяют *CEx03aView* — главный класс приложения. Объект класса *CEx03aView* соответствует рабочему окну программы, где и происходят все «события».

2. **Выполните компиляцию и компоновку сгенерированного кода.** Помимо генерации кода, MFC Application Wizard создает для вашего приложения файлы проекта и рабочего пространства. Файл проекта Ex03a.vsproj описывает все зависимости файлов, а также параметры компилятора и компоновщика. Так как новый проект становится текущим проектом Visual C++ .NET, вы можете собрать приложение, выбрав Build Ex03a.exe из меню Build или щелкнув кнопку Build на панели инструментов:



Если сборка прошла успешно, в подкаталоге Debug каталога \vcppNet\ex03a создается исполняемый файл ex03a.exe. Файлы OBJ и другие промежуточные файлы также помещаются в каталог Debug. Сравните структуру каталогов на диске со структурой страницы Solution Explorer.

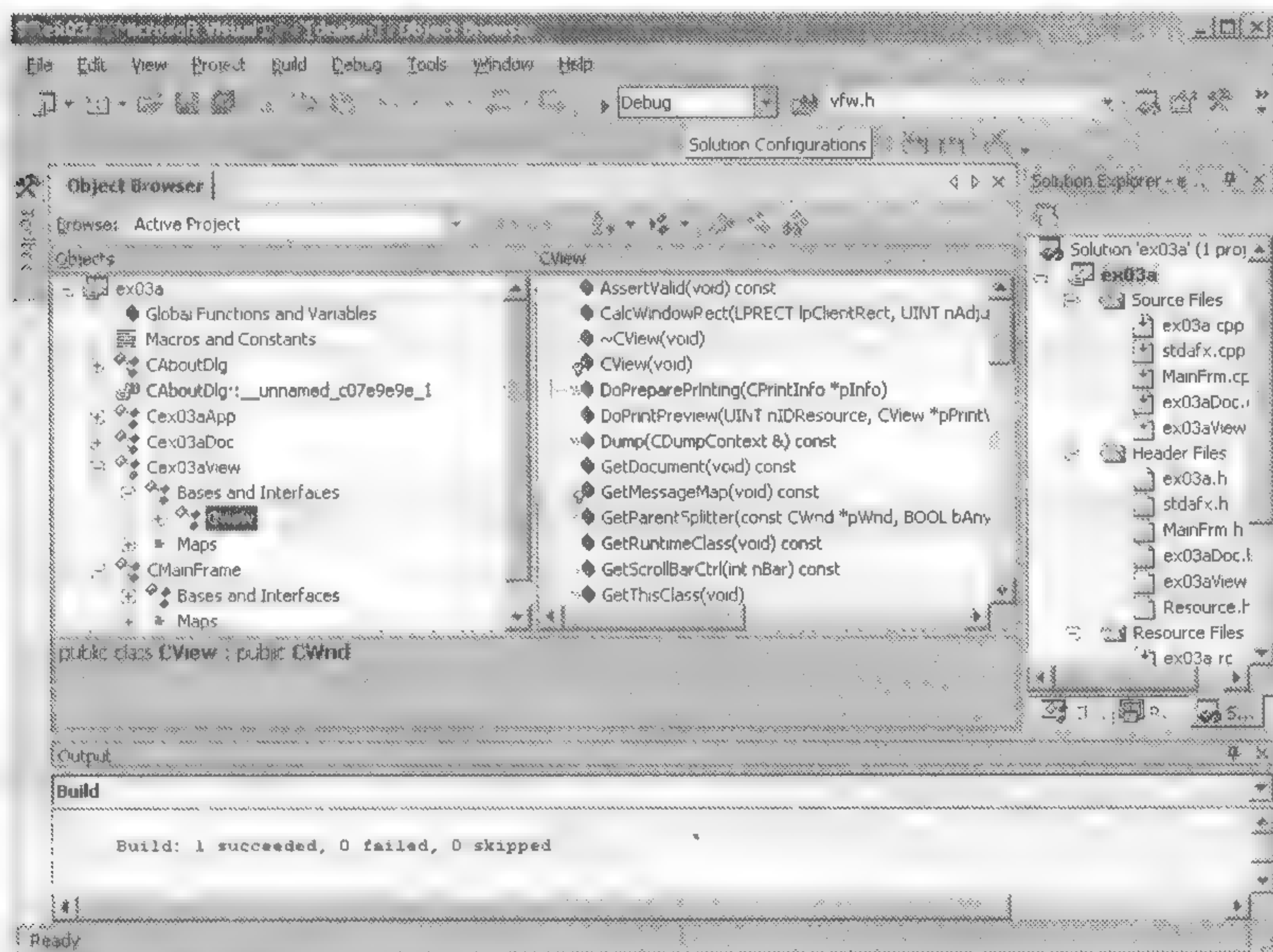


Solution Explorer представляет логическую структуру проекта. Заголовочные файлы располагаются в разделе Header Files, хотя физически они хранятся в том же подкаталоге, что и файлы CPP. Файлы ресурсов хранятся в подкаталоге \res.

3. **Протестируйте полученное приложение.** В меню Debug выберите команду Start Without Debugging. Поэкспериментируйте с программой. Она мало на что способна, да? (А что вы хотели, не написав ни строчки кода?) На самом деле, как вы, вероятно, догадываетесь, у программы много возможностей — просто они еще не активизированы. Закончив эксперименты, закройте окно программы.

4. **Просмотрите исходные тексты программы.** Нажмите CTRL+ALT+J, чтобы открыть окно Object Browser. Если параметры проекта не требуют создания базы данных средства просмотра, Visual C++ .NET предложит изменить их и перекомпилировать программу. [Чтобы изменить параметры самостоятельно, выберите Properties в меню Project. Перейдите к папке C/C++, щелкните значок Browse Information измените значение свойства Enable Browse Information на All Browse Information (/FR).]

   Раскрыв ветви иерархии, вы должны получить результат, аналогичный представленному ниже.



Сравните эти результаты с содержимым страницы Class View:



Class View показывает иерархию классов почти так же, как Object Browser, но зато последний показывает все имеющиеся функции класса, а Class View — только переопределенные. Если вам хватает Class View, не трудитесь создавать базу данных браузера.

Класс *CEx03aView*

Класс *CEx03aView* сгенерирован MFC Application Wizard специально для приложения Ex03a. (Мастер создает имена классов на основании имени проекта, заданного в его первом диалоговом окне.) *CEx03aView* находится в конце длинной цепочки наследования классов библиотеки MFC, как видно в окне Object Browser. В классе собраны функции-члены и переменные-члены со всей цепочки. Сведе-

ния об этих классах см. в справочнике *Microsoft Foundation Class Reference* (в интерактивном или бумажном варианте), но обязательно просматривайте описания всех базовых классов, так как описания унаследованных функций-членов обычно не повторяются в производных классах.

Важнейшие базовые классы вида *CEx03aView* — *CWnd* и *CView*. *CWnd* придает *CEx03aView* свойства окна, а *CView* обеспечивает связь с остальными частями каркаса приложения, в частности, с документом и рамочным окном, как вы увидите в главе 12.

Рисование внутри окна представления: Windows GDI

Теперь вы готовы к созданию кода, который будет рисовать в окне представления. Вы внесете несколько изменений прямо в исходный текст Ex03a. Конкретнее, вам понадобится наполнить реализацию *OnDraw* в Ex03aView.cpp и поработать с контекстом устройства и интерфейсом GDI.

Функция-член *OnDraw*

OnDraw — это виртуальная функция-член класса *CView*, которую каркас приложения вызывает всякий раз, когда нужно перерисовать окно представления. Перерисовка требуется, когда пользователь изменил размеры окна или открыл ранее невидимые его части, либо само приложение изменило данные окна. В первых двух случаях *OnDraw* вызывается каркасом приложения автоматически; однако если данные окна изменены функцией изнутри программы, эта функция должна уведомить Windows об изменениях, вызвав унаследованную классом «вид» функцию-член *Invalidate* (или *InvalidateRect*). Код *Invalidate* впоследствии вызовет *OnDraw*.

Хотя внутри окна разрешается рисовать в любой момент времени, рекомендуется все же накапливать изменения и обрабатывать их «одним махом», вызвав функцию *OnDraw* лишь раз, — тогда программа сможет реагировать как на события, сгенерированные ею самой, так и на события, инициированные Windows, например, на изменения размеров окна.

Контекст устройства в Windows

Как вы помните из главы 1, Windows не допускает прямого доступа к аппаратуре дисплея, а взаимодействует с ней через уровень абстрагирования под названием *контекст устройства* (device context), связанный с окном. В библиотеке MFC контекст устройства представлен объектом класса C++ с именем *CDC*, который передается в *OnDraw* по ссылке (как указатель). Указатель на *CDC* позволяет задействовать множество функций этого класса для рисования.

Добавление кода рисования в программу Ex03a

А теперь напишем код, отображающий в окне представления текст и круг. Убедитесь, что проект Ex03a открыт в Visual C++ .NET. Чтобы найти функцию, можно воспользоваться Class View (дважды щелкните *OnDraw*) либо открыть исходный файл ex03aView.cpp из Solution Explorer и отыскать функцию в тексте.

1. **Отредактируйте функцию *OnDraw* в ex03aView.cpp.** Найдите в файле Ex03aView.cpp функцию *OnDraw*, сгенерированную MFC Application Wizard:

```
void CEx03aView::OnDraw(CDC* /* pDC */)
{
    CEx03aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}
```

Удалите символы комментария с указателя на контекст устройства и замените код на текст, выделенный полужирным шрифтом:

```
void CEx03aView::OnDraw(CDC* pDC)
{
    pDC->TextOut(0, 0, "Hello, world!");    // Вывод шрифтом по умолчанию
                                            // в левом верхнем углу
    pDC->SelectStockObject(GRAY_BRUSH);     // Выбрать кисть для заполнения круга
    pDC->Ellipse(CRect(0, 20, 100, 120));   // Нарисовать серый круг
                                            // диаметром 100 единиц
}
```

Вызов *GetDocument* можно спокойно удалить, поскольку пока мы не работаем с документами. Функции *TextOut, SelectStockObject* и *Ellipse* — члены класса контекста устройства *CDC* каркаса приложения. Функция *Ellipse* рисует круг, если длина ограничивающего прямоугольника равна его ширине.

Для тех, кто программирует в Win32

Не сомневайтесь, стандартная функция *WinMain* и оконные процедуры скрыты в каркасе приложения. Вы увидите эти функции, когда мы будем обсуждать классы библиотеки MFC для окна-рамки и приложения. Сейчас вас, возможно, удивляет, куда же делось сообщение *WM_PAINT*? Казалось бы, при обработке данного сообщения должно выполняться рисование в окне, а контекст устройства надо получать из структуры *PAINTSTRUCT*, возвращаемой функцией Windows *BeginPaint.*

Оказывается, всю эту черновую работу за вас выполнил каркас приложения и передал контекст устройства (в форме указателя на объект) виртуальной функции *OnDraw.* Как говорилось в главе 2, настоящие виртуальные функции в оконных классах MFC довольно редки. Большинство сообщений Windows каркас приложения направляет на обработку функциям карты сообщений. Программисты, работавшие с MFC 1.0, всегда создавали для своих производных классов-окон функцию таблицы сообщений *OnPaint.* Начиная же с версии 2.5, функция *OnPaint* находится в таблице сообщений класса *CView* и выполняет полиморфный вызов функции *OnDraw.* Почему? Потому что *OnDraw* должна поддерживать не только дисплей, но и принтер. Как *OnPaint*, так и *OnPrint* вызывают функцию *OnDraw*, что позволяет использовать один и тот же код рисования и для принтера, и для дисплея.

Для работы с прямоугольниками Windows библиотека MFC предоставляет удобный класс *CRect*. Временный объект *CRect* — аргумент, ограничивающий прямоугольник для функции рисования эллипса. Класс *CRect* будет часто встречаться в примерах этой книги.

2. **Скомпилируйте и оттестируйте Ex03a.** Выберите команду Build из меню Build и, если нет ошибок компиляции, снова запустите программу. Теперь видно, что ваша программа кое-что да умеет!

Первое знакомство с редакторами ресурсов

Теперь у нас есть готовая программа — самое время познакомиться с редакторами ресурсов. Хотя файл ресурсов приложения Ex03a.rc — текстовый ASCII-файл, изменять его в текстовом редакторе — не лучшая идея: для этого существуют редакторы ресурсов.

Что содержит файл Ex03a.rc

Файл ресурсов во многом определяет внешний вид и поведение приложения Ex03a. Он содержит (или указывает на) ресурсы Windows (табл. 3-2).

Табл. 3-2. Ресурсы Windows в MFC-приложении

Ресурс	Описание
«Быстрая клавиша» (Accelerator)	Задает клавиши, нажатие на которые эквивалентно выбору элементов меню и кнопок панели управления.
Диалоговое окно (Dialog)	Определяет формат и содержимое диалоговых окон. В Ex03a есть только диалоговое окно About («О программе»).
Значок (Icon)	Значки (версии 16×16 и 32×32 точки), аналогичные значкам приложений, которые отображаются в Проводнике (Windows Explorer) и в диалоговом окне About приложения. В качестве значка приложения Ex03a используется логотип MFC.
Манифест (Manifest)	Содержит информацию о типах времени исполнения для приложения.
Меню (Menu)	Меню верхнего уровня приложения и связанные с ним раскрывающиеся меню.
Таблица строк (String table)	Строки, не являющиеся частью исходного кода C++.
Панель инструментов (Toolbar)	Ряд кнопок непосредственно под меню.
Версия (Version)	Описание программы, номер ее версии, язык и т. д.

Кроме перечисленных ресурсов, ex03a.rc содержит операторы:

```
#include "afxres.h"
#include "afxres.rc"
```

Они подключают некоторые ресурсы библиотеки MFC, общие для всех приложений. В их числе строки, графические кнопки и элементы, необходимые для печати и OLE.

Примечание Если вы используете MFC в виде совместно используемой DLL-библиотеки, общие ресурсы хранятся в самой DLL MFC.

Файл ex03a.rc также содержит оператор:

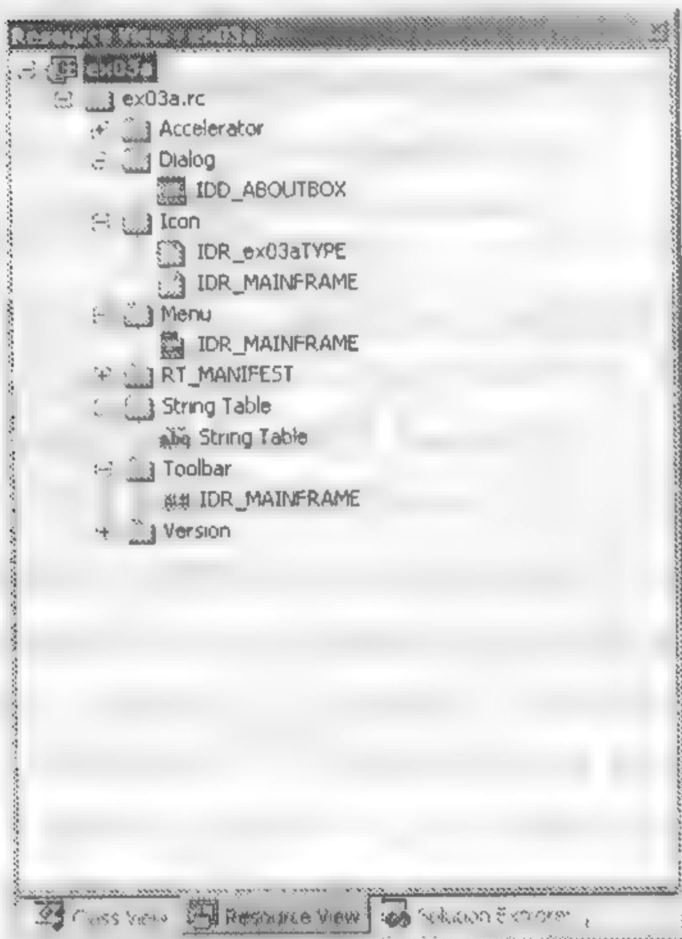
```
#include "resource.h"
```

который вводит в приложение три константы *#define*: *IDR_MAINFRAME* (определяет меню, значок, таблицы строк и «быстрых клавиш»), *IDR_EX03ATYPE* (определяет значок документа по умолчанию, однако в этой программе мы его не используем) и *IDD_ABOUTBOX* (идентификатор диалогового окна About). Этот же файл resource.h неявно включается исходными файлами приложения. Если средствами редактора ресурсов добавить другие константы (символы), то их определения в конечном счете попадут в resource.h. Будьте внимательны, редактируя этот файл в текстовом редакторе: внесенные вами изменения могут быть удалены, когда вы в следующий раз используете редактор ресурсов.

Работа с редактором диалоговых окон

Редактор диалоговых окон служит для создания и редактирования ресурсов диалоговых окон.

1. **Откройте RC-файл проекта.** В меню View выберите команду Resource View. Раскройте узлы — вы должны увидеть следующее:



2. **Изучите ресурсы приложения.** Если выбрать ресурс двойным щулчком, открывается другое окно с набором инструментов, соответствующих данному ресурсу. Открыв ресурс диалогового окна, вы увидите палитру элементов управления. Если она не появилась, выберите в меню View команду Toolbox.
3. **Измените диалоговое окно IDD_ABOUTBOX.** Внесите небольшие изменения в представленное ниже диалоговое окно About.

   Вы можете изменить размер окна, двигая мышью его правый и левый края, переместить кнопку OK, изменить текст и т. д. Просто выделите элемент щелчком, а затем, щелкнув правой кнопкой, измените его свойства.

4. **Заново соберите проект с измененным файлом ресурсов.** В Visual C++ .NET в меню Build выберите команду Build Ex03a.exe. Кстати, перекомпилировать код C++ не понадобится. Visual C++ .NET сохранит отредактированный файл ресурсов, затем компилятор ресурсов (rc.exe) обработает Ex03a.rc и создаст скомпилированный вариант Ex03a.res, который передается компоновщику. Последний выполнит свою задачу быстро, так как в данном случае ему достаточно скомпоновать лишь изменения.
5. **Протестируйте новую версию приложения.** Снова запустите программу Ex03a, в меню Help выберите команду About и убедитесь, что ваши изменения видны в диалоговом окне.

Конфигурации Debug и Release

При сборке приложения вы вправе выбрать один из двух вариантов конфигурации приложения: отладочный (Debug) или окончательный (Release). При генерации нового проекта MFC Application Wizard создает одну из конфигураций (табл. 3-3).

Табл. 3-3. Параметры, определяемые MFC Application Wizard по умолчанию

Параметр	Окончательная сборка (Release)	Отладочная сборка (Debug)
Отладка по исходному тексту	Отключена	Включена для компилятора и компоновщика
Диагностические макросы MFC	Отключены (определен *NDEBUG*)	Включены (определен *_DEBUG*)
Библиотеки	Рабочие библиотеки MFC	Отладочные библиотеки MFC
Оптимизация при компиляции	Оптимизация по скорости (в Learning Edition отсутствует)	Без оптимизации (ускоренная компиляция)

Разработка приложений ведется в *режиме отладочной сборки* (Debug mode), а перед поставкой программа собирается заново в *режиме окончательной сборки* (Release mode). Исполняемые файлы, собранные в режиме Release, характеризуются меньшим размером и работают быстрее. Текущая конфигурация выбирается из списка в окне Build (рис. 1-2 в главе 1). По умолчанию результаты и промежуточные файлы сборки проекта в отладочном режиме, хранятся в подпапке Debug, а файлы для окончательной сборки — в подпапке Release. Вы вправе задать другие каталоги на странице свойств General папки Configuration Properties, доступной в диалоговом окне свойств проекта.

Вы можете создавать собственные конфигурации, выбрав в меню Build команду Configuration Manager.

Предкомпилированные заголовочные файлы

При создании проекта MFC Application Wizard генерирует параметры и файлы, необходимые для предварительной компиляции заголовочных файлов. Для эффективного управления проектами нужно знать, как работают *предкомпилированные заголовочные файлы* (precompiled headers).

Примечание В Visual C++ .NET два «режима» предкомпиляции заголовочных файлов: автоматический и ручной. При *автоматической* (automatic), активизируемой параметром командной строки компилятора */Yx*, результаты работы компилятора сохраняются в файле «базы данных». Предкомпиляция *вручную* (manual) активизируется параметрами компилятора */Yc* и */Yu*. Созданные при этом заголовочные файлы применяются в проектах, сгенерированных MFC Application Wizard.

Предкомпилированные заголовочные файлы представляют собой «моментальные снимки», которые делает компилятор на определенной строке исходного текста. В MFC-программах такой снимок обычно делается сразу после оператора:

```
#include "stdafx.h"
```

Файл StdAfx.h содержит операторы *#include* для заголовочных файлов библиотеки MFC. Содержимое файла зависит от выбранного режима MFC Application Wizard, однако в нем *всегда* есть операторы:

```
#include <afxwin.h>
#include <afxext.h>
```

Если в приложении применяются составные документы, StdAfx.h также содержит строку:

```
#include <afxole.h>
```

А если вы работаете с Automation или элементами управления ActiveX, он содержит:

```
#include <afxdisp.h>
```

Если же вы используете Internet Explorer 4 Common Controls, то StdAfx.h содержит

```
#include <afxdtctl.h>
```

Вам могут потребоваться и другие заголовочные файлы. Так, заголовочный файл для классов наборов на основе шаблонов подключается оператором:

```
#include <afxtempl.h>
```

В файле StdAfx.cpp содержится единственный оператор:

```
#include "StdAfx.h"
```

Он-то и используется для генерации файла предкомпилированных заголовочных файлов. Заголовочные файлы библиотеки MFC, включаемые в файле StdAfx.h, никогда не изменяются, но их компиляция требует времени. Параметр компилятора */Yc*, используемый только для StdAfx.cpp, вызывает создание предкомпилированного заголовочного файла (с расширением .pch). Параметр */Yu*, применяемый для остальных исходных файлов, вызывает использование существующего PCH-файла. Для определения имен PCH-файла применяется параметр */Fp*. В отсутствие этого параметра в подкаталоге результатов текущей конфигурации создается файл с именем проекта и расширением PCH. Весь процесс показан на рис. 3-1.

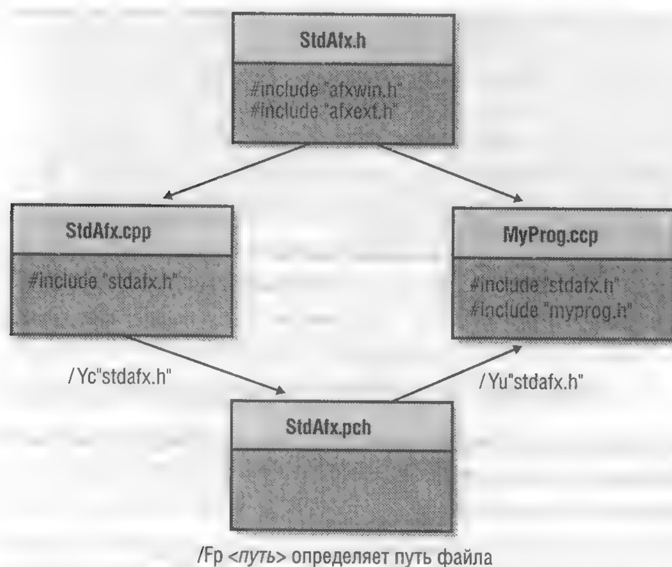


Рис. 3-1. *Процесс генерации предкомпилированных заголовков*

AppWizard автоматически генерирует параметры */Yc* и */Yu*, но вы вправе внести коррективы. Можно задавать параметры компилятора для отдельных исходных файлов. Если в диалоговом окне свойств проекта (Property Pages) на странице Precompiled Headers папки C/C++ выбрать только файл StdAfx.h, вы увидите параметр */Yc*, который переопределят параметр */Yu*, заданный для всего проекта.

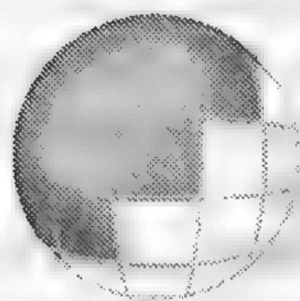
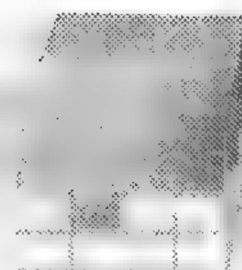
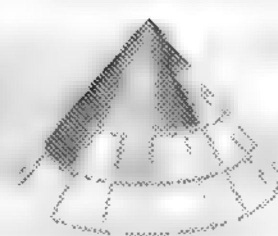
Учтите: PCH-файлы достаточно объемны — в среднем около 10 Мб. Если с ними обращаться неосмотрительно, жесткий диск может быстро переполниться. Рекомендуется периодически очищать каталоги Debug своих проектов или же размещать все PCH-файлы в одном каталоге, определяя параметр компилятора */Fp*.

Два способа запуска программы

Visual C++ .NET позволяет запускать программу непосредственно (нажав клавиши Ctrl+F5) или в отладчике (клавиша F5). Непосредственный запуск выполняется гораздо быстрее, так как Visual C++ .NET не требуется предварительно загружать отладчик. Если вам не нужны трассировочные сообщения и точки останова, запускайте программу с помощью Ctrl+F5.

ГЛАВА

4

Мастера Visual C++ .NET

При разработке программ для операционных систем Microsoft приходится писать много шаблонного кода. На заре Windows большинство программистов начинало разработку Windows-приложения, вооружившись лишь книгой Чарльза Петцольда (Charles Petzold) «Programming Windows» и комплектом разработчика Windows Software Development Kit (SDK). Даже в документации Windows SDK рекомендовался метод разработки с широким использованием унаследованного кода.

Чтобы разобраться с основами любой технологии, надо самостоятельно написать весь код приложения. Но наступает момент, когда написание однообразного шаблонного кода перестает быть упражнением и превращается в рутину и пустую трату времени. В Microsoft Visual Studio .NET эту проблему решает встроенный набор генераторов кода, облегчающих создание проектов любого типа. Доступные шаблоны проектов отображаются в окне New Project, открывающемся при последовательном выборе команд New и Project в меню File. Вам достаточно выбрать шаблон проекта, пробежать несколько диалоговых окон конфигурирования проекта и щелкнуть кнопку Finish. Вуаля — у вас работающее приложение!

Но это еще не все. Технология мастеров открыта — вы вправе писать собственные мастера. В этой главе вы узнаете, как это делается в Visual Studio .NET.

Типы мастеров

В Visual Studio .NET два типа мастеров: с пользовательским интерфейсом и без — все определяется сложностью мастера и желанием автора. Большинство описанных в этой книге мастеров относится к первому типу. Например, MFC Application Wizard состоит из нескольких страниц, на которых определяются такие параметры, как тип интерфейса (SDI или MDI), необходимость поддержки печати и предварительного просмотра, а также применения элементов управления ActiveX. В некоторых простых приложениях пользовательский интерфейс не нужен.

В мастерах без пользовательского интерфейса достаточно определить название проекта — файлы проекта создаются в соответствии с выбранными шаблонами. Мастера с пользовательским интерфейсом более интерактивны и часто состоят из нескольких страниц.

В сущности исходный код всех мастеров Visual C++ .NET доступен — его вы найдете в каталоге \Program Files\Microsoft Visual Studio .NET\VC7\VCWizards.

Как работают мастера

Сначала мы узнаем, как работают мастера, и познакомимся с тремя основными компонентами мастера: исходным шаблонным кодом, пользовательским интерфейсом и результирующим (сгенерированным) кодом.

Главная задача генератора кода — избавить вас от написания шаблонного кода. Это может быть вполне рабочий и поддающийся компиляции «скелет» приложения или библиотеки. Однако код должен решать основную задачу проекта. Например, при написании приложения, которое создает платежные ведомости, имена классов должны иметь внятные названия, скажем, *CPayrollDoc* (документ платежной ведомости), *CPayrollView* (просмотр платежной ведомости) или *CPayrollFrame* (окно-рамка платежной ведомости). В числе прочего в обязанности мастера вменяется присвоение шаблонным классам простых понятных имен, вводимых разработчиком.

Мастер позволяет добавлять или отбрасывать отдельные части исходного кода. Так, если на странице мастера установить флажок диалогового окна About, мастер добавит исправленный код соответствующего окна в конечное приложение.

Выбор вариантов реализуется в пользовательском интерфейсе мастера. Сердце интерфейса мастера — HTML-элемент управления *IVCWizCtrlUI*. По сути пользовательский интерфейс мастеров Visual Studio .NET написан на языке HTML. Во время работы мастера *IVCWizCtrlUI* находит и отображает файлы пользовательского интерфейса в окне мастера. Мастер отвечает за навигацию по страницам и генерацию кода по щелчку кнопки Finish.

Число страниц мастера не ограничивается, причем каждая страница — это отдельный HTML-файл. Для перемещения между страницами мастера служат кнопки Next и Back (впрочем, вы вправе организовать и иной порядок перемещения). HTML-файлы пользовательского интерфейса содержат тэг *SYMBOL*, который описывает значения по умолчанию для определяемых разработчиком параметров.

Мастер поддерживает таблицу символов на протяжении всего времени своей работы. Таблица символов служит для выполнения подстановок. Объявленные в HTML-файле символы по щелчку кнопки Finish записываются в таблицу символов. Вот пример HTML-кода мастера:

```
<SYMBOL NAME='SOURCE_FILE' VALUE='MySource.cpp' TYPE=text></SYMBOL>
```

В пользовательском интерфейсе мастера текстовое окно служит для ввода информации пользователем. Идентифицируется текстовое окно символом *SOURCE_FILE*. Это ключевой символ, который мастер применяет при подстановке в исходных файлах. Сейчас вы узнаете, как это работает. По сути каждый HTML-файл мастера записывает выбранные пользователем варианты в таблицу символов.

Внутренняя логика мастера обычно базируется на JScript. При необходимости особого поведения мастера для доступа к модели Visual C++ Wizard можно использовать функции JScript, которые размещаются на HTML-странице в разделе *<SCRIPT LANGUAGE='JSCRIPT'>*.

Примечание Подробнее о модели мастеров Visual C++ Wizard и других объектных моделях, составляющих расширяемую объектную модель Visual C++ Extensibility Object Model, см. библиотеку MSDN.

Создание мастера

Первый шаг при создании мастера — написание и отладка шаблонного приложения. Затем средствами Visual C++ .NET создается «чистый» мастер. В состав Visual Studio .NET включен мастер Custom Wizard, применяемый для создания мастеров. Он генерирует все файлы, необходимые для реализации мастера.

Чтобы создать мастер, последовательно выберите команды New и Project в меню File, в открывшемся диалоговом окне выберите папку Visual C++ Projects и шаблон Custom Wizard. Введите имя мастера в поле Name. Custom Wizard состоит из двух страниц: обзорной (Overview) и страницы параметров Application Settings. Последняя позволяет определить понятное имя и количество страниц в мастере, а также указать, должен ли у мастера быть пользовательский интерфейс. Мастер Custom Wizard создает несколько файлов (табл. 4-1).

Табл. 4-1. Файлы, создаваемые мастером Custom Wizard

Файлы	Описание
Project.vsz	Тестовый файл ядра мастера. Предоставляет информацию о контекстных и дополнительных (необязательных) параметрах.
Project.vsdir	Тестовый файл сервиса маршрутизации между оболочкой Visual Studio и элементами в проекте мастера.
HTML-файлы (при необходимости)	Файлы, реализующие пользовательский интерфейс мастера. Для мастеров без пользовательского интерфейса HTML-файлы не нужны.
	Если мастер состоит из одной страницы, создается файл Default.htm. В противном случае дополнительные страницы называются Page_*<номер_страницы>*.htm.
Файлы сценариев	Логика работы мастера заключена в сценариях. Для каждого проекта в мастере создаются файлы JScript с именами Default.js и Common.js. Они содержат JScript-функции, применяемые для доступа к моделям Visual C++ Wizard, Code, Project, и Resource Editor и тонкой настройки мастера. Добавляют и настраивают функции в файле проекта мастера Default.js.
Шаблонные файлы	Набор текстовых файлов с директивами в каталоге Templates. Файлы анализируются и вставляются в таблицу символов согласно выбранным пользователем параметрам. Соответствующую информацию получают путем прямого доступа к таблице символов относящегося к мастеру элемента управления.

см. след. стр.

Табл. 4-1. *(продолжение)*

Файлы	Описание
Templates.inf	Текстовый файл со списком всех относящихся к проекту шаблонов.
Default.vcproj	XML-файл, содержащий сведения о типе проекта
Sample.txt	Шаблонный файл, показывающий, как использовать директивы мастера.
ReadMe.txt	Шаблонный файл, содержащий информацию обо всех файлах, созданных мастером Custom Wizard.
Файлы изображений (при необходимости)	Файлы изображений — значки, GIF-файлы, BMP-растры и другие поддерживаемые в HTML форматы. Служат для «украшения» интерфейса мастера. Понятно, что в мастере без пользовательского интерфейса файлов изображений не нужно.
Styles.css (при необходимости)	Файл, определяющий стили пользовательского интерфейса. Если пользовательского интерфейса нет, мастер Custom Wizard не создает CSS-файл.
Common.js	Набор JScript-функций, используемый всеми мастерами. На самом деле этот файл мастером Custom Wizard не создается — он просто добавляется в результирующий код.

Создание мастера для разработки Web-приложений на управляемом C++

Сейчас вы научитесь создавать мастер приложения генерирующий Web-приложение с применением ASP.NET и управляемого C++. Подробнее о написании приложений на основе Web Forms с использованием ASP.NET и управляемого C++ вы узнаете во второй части книги. А пока мы создадим мастер приложения, генерирующий приложение на основе Web Forms. При этом надо создать ряд файлов, а также неплохо предусмотреть трассировочные и отладочные сообщения, несколько управляющих элементов разных типов, чтобы наглядно познакомиться с работой мастера. Для приложения на основе Web Forms следует создать несколько файлов: файлы исходного кода DLL-библиотеки на управляемом C++, файлы ASP.NET (ASPX), файл Web.Config и файл решения Visual Studio. В каждом из этих файлов надо предусмотреть возможность подстановок, выполняемых мастером приложения после получения входных данных от пользователя.

Мы создадим мастер с помощью Custom Wizard и назовем его *ManagedCWebFormWizard*. Чтобы процесс был понятнее, мы будем создавать мастер с одной страницей, но в своем мастере вы вправе создать сколько угодно страниц.

Сам пользовательский интерфейс будет содержать флажки, позволяющие добавлять элементы управления, включать/отключать отладочные и трассировочные сообщения. Solution Explorer позволяет посмотреть на HTML-страницу пользовательского интерфейса мастера. Редактирование этой страницы похоже на редактирование стандартных диалоговых окон: элемент управления выбирается на инструментальной панели Toolbox в левой части окна Visual Studio .NET и переносится на страницу, затем ему присваиваются свойства в окне Properties. На странице мастера 6 флажков. Три флажка управляют добавлением трех элементов управления в Web Form: флажка (CheckBox), надписи (Label) и текстового поля

(TextBox). В окне Properties определяются названия каждого из элементов. Флажок называется *UseTextBox*, надпись — *UseLabel*, а текстовое поле — *UseCheckBox*. Во время генерации кода мастер отыскивает эти символы и добавляет соответствующий код в ASPX-файл и текст страницы.

Остальные три флажка позволяют управлять отладкой: первый служит для трассировки страницы, другой — для трассировки по запросу, третий — для включения отладки. Идентификаторы флажков — *UsePageTracing*, *UseRequestTracing* и *UsePageDebugging*. Как и в случае со страницей пользовательского интерфейса, мастер ищет эти символы и добавляет соответствующий код в создаваемый проект.

На рис. 4-1 показана страница мастера — файл Default.htm — в действии.



Рис. 4-1. *Страница пользовательского интерфейса мастера* ManagedCWebFormWizard

Размещенные на странице элементы управления следует сопоставить с символами, которые мастер будет применять для подстановок. Исходная страница пользовательского интерфейса мастера (default.htm) содержит группу записей о символах. Измените символы мастера Web-приложения так:

```
<SYMBOL NAME="UseCheckBox" TYPE="checkbox" VALUE="false"></SYMBOL>
<SYMBOL NAME="UseTextBox" TYPE="checkbox" VALUE="false"></SYMBOL>
<SYMBOL NAME="UseLabel" TYPE="checkbox" VALUE="false"></SYMBOL>
<SYMBOL NAME="UsePageTracing" TYPE="checkbox" VALUE="false"></SYMBOL>
<SYMBOL NAME="UseRequestTracing" TYPE="checkbox" VALUE="false"></SYMBOL>
<SYMBOL NAME="UsePageDebugging" TYPE="checkbox" VALUE="false"></SYMBOL>
```

Заметьте: символы сопоставляются отдельным флажками на странице пользовательского интерфейса мастера.

Далее надо взять исходный код и вставить примечания с тех местах, где мастер должен добавлять другой код. Готовый шаблонный код хранится в каталоге Templates мастера. В конечном варианте *ManagedCWebForm* должен содержать три файла: заголовочный файл с классом C++, ASPX-файл с информацией о разметке Web-страницы и файл Web.Config с конфигурационными параметрами. Шаблон-

ные коды всех этих файлов будут располагаться в каталоге Templates мастера. Познакомимся с шаблонным кодом, используемым мастером для создания приложения. Вот код заголовочного файла C++:

```
// ManagedCWebForm.h

#pragma once

using namespace System;
#using <System.Dll>
#using <System.Web.dll>

using namespace System;
using namespace System::Web;
using namespace System::Web::UI;
using namespace System::Web::UI::WebControls;
using namespace System::Collections;
using namespace System::ComponentModel;

namespace ProgVSNET_ManagedCWebForm
{
   public __gc class ManagedCWebPage : public Page
   {
      public:
         Button* m_button;

[!if UseLabel]
         Label* m_label;
[!endif]
[!if UseTextBox]
         TextBox* m_text;
[!endif] [!if UseCheckBox]
         CheckBox* m_check;
[!endif]

      ManagedCWebPage()
      {
         // Здесь вставляется код конструктора...
      }

      void SubmitEntry(Object* o, EventArgs* e)
      {
         // Вызывается по щелчку кнопки Submit
         // Здесь вставляется код, исполняемый при загрузке страницы...
         String* str;

         str = new String("Hello ");
         str = str->Concat(str, m_text->get_Text());
         str = str->Concat(str, new String(" you pushed Submit"));
[!if UseLabel]
         m_label->set_Text(str);
```

```
[!if UseLabel]
      }

      void Page_Load(Object* o, EventArgs* e)
      {
         // Здесь вставляется код, исполняемый при загрузке страницы...
[!if UsePageTracing]
         Trace->Write("Custom", "Inside Page_Load");
[!endif]
         if(!IsPostBack) {
         }
      }
   };
}
```

При генерации кода мастер ищет ключевой символ, заключенный в квадратные скобки, и проверяет его наличие в таблице символов. В нашем примере это простые булевы выражения. Если флажки установлены, значит, надо включить соответствующие элементы управления или отладочные сообщения. Иначе соответствующий код следует «выкинуть» из создаваемого кода. Аналогично следует обработать все создаваемые файлы. Например, мастер возьмет следующий шаблонный код страницы ASP.NET, проверит вхождение символов *UseRequestTracing*, *UseTextBox*, *UseLabel* и *UseCheckBox* и решит, какой код присоединить:

```
<%@ Page Language="C#"
[!if UseRequestTracing]
   Trace=true
[!endif]
   Inherits="ProgVSNET_ManagedCWebForm.ManagedCWebPage"
%>

<html>
<body>
<form runat=server>
<h2>ASP.NET Web Form</h2>

<br><br><br>

   <asp:Button Text="Sumit Entry" id="m_button"
      OnClick="SubmitEntry" runat=server /><br/>

   <asp:Label Text="Type your name here"  runat=server />


[!if UseTextBox]
   <asp:TextBox id="m_text" runat=server /><br/>
[!endif]
[!if UseCheckBox]
   <asp:CheckBox id="m_check" runat=server /> <br/>
[!end]
```

```
[!if UseLabel]
   <asp:Label id="m_label" runat=server />
[!endif]

</form>
</body>
</html>
```

Последним создается файл Web.Config — XML-файл, используемый в ASP.NET для конфигурирования Web-приложения. Здесь трассировка и отладка на уровне страницы включаются/выключаются в зависимости от состояния флажков:

```
<configuration>
   <system.web>
[!if UsePageDebugging]
   <compilation debug='true'></compilation>
[!endif]
[!if UsePageTracing]
   <trace enabled='true'></trace>
[!endif]
   </system.web>
</configuration>
```

Кроме шаблонного кода, мастер также должен знать, какие файлы добавлять в создаваемый проект приложения. Каталог Templates содержит файл Templates.inf со списком создаваемых файлов. Файл Templates.inf указывает мастеру, какие файлы должны содержаться в конечном проекте. Мы добавим в него записи о файлах ManagedCWebForm.cpp, ManagedCWebForm.h, ManagedCWebForm.aspx и Web.config. Этот файл работает так же, как и описанные до этого: мастер проверяет символы в таблице символов и генерирует приложение в зависимости от выбора, сделанного пользователем на странице интерфейса. В процессе создания проекта код сценария вызывает функцию *GetTargetName*, чтобы изменить имена базовых файлов (ManagedCWebForm.aspx, ManagedCWebForm.cpp и ManagedWebForm.h) в соответствии с именем проекта, заданным пользователем при запуске мастера. Здесь показан модифицированный метод GetTargetName, который заменяет имена файлов.

```
function GetTargetName(strName, strProjectName)
{
   try
   {
      var strTarget = strName;

      if (strName.substr(0, 15) == "ManagedCWebForm")
      {
         var strlen = strName.length;
         strTarget = strProjectName + strName.substr(15, strlen - 15);
      }
      return strTarget;
   }
   catch(e)
   {
```

```
        throw e;
    }
}
```

После создания файлов мастер на их основании создает проект. Сценарии создания проекта размещены в подкаталоге сценариев проекта мастера. Исходный сценарий, сгенерированный мастером Custom Wizard, содержит метод *AddConfig*. Объектная модель проектов в Visual Studio .NET позволяет изменять конфигурацию сгенерированного проекта. Далее следует исходный код, который устанавливает переключатель DLL-библиотеки и генерирует управляемую сборку. (Об управляемом коде см. часть 6).

```
function AddConfig(proj, strProjectName)
{
    try
    {
        var config = proj.Object.Configurations('Debug');
        config.IntermediateDirectory = 'Debug';
        config.OutputDirectory = 'Debug';

        config.ConfigurationType = typeDynamicLibrary;
        var CLTool = config.Tools('VCCLCompilerTool');
        // TODO: Add compiler settings
        CLTool.CompileAsManaged = managedAssembly;

        var LinkTool = config.Tools('VCLinkerTool');
        // TODO: Add linker settings

        config = proj.Object.Configurations('Release');
        config.IntermediateDirectory = 'Release';
        config.OutputDirectory = 'Release';

        var CLTool = config.Tools('VCCLCompilerTool');
        // TODO: Add compiler settings
        CLTool.CompileAsManaged = managedAssembly;

        var LinkTool = config.Tools('VCLinkerTool');
        // TODO: Add linker settings
    }
    catch(e)
    {
        throw e;
    }
}
```

Мы должны позаботиться о том, чтобы Visual Studio .NET «узнала» о существовании созданного мастера. Для этого нужно разместить его в отдельном подкаталоге каталога \Program Files\Microsoft Visual Studio .NET\VC7\VCWizards. Файлы пользовательского интерфейса размещаются в подкаталоге HTML, файлы шаблонного кода — в подкаталоге Templates, файлы изображений — в подкаталоге Images, файлы сценариев — в подкаталоге Scripts. Все эти подкаталоги хранятся внутри

(на уровень ниже) каталога мастера. Файлы пользовательского интерфейса и шаблонные файлы можно локализовать. Файлы VSDIR, VSZ и значок размещаются в каталоге \Program Files\Microsoft Visual Studio .NET\VC7\VCProjects. Как уже говорилось, файлы VSDIR и VSZ генерирует мастер Custom Wizard.

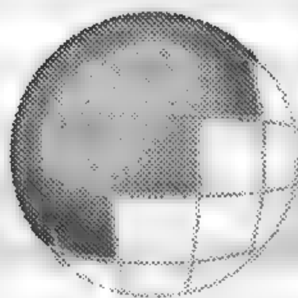
Модель мастера приложений в Visual Studio .NET достаточно гибка и богата функциями. Мы рассмотрели только подстановку, определяемую состоянием флажков. Существует множество других путей создания мастеров приложений для генерации самых разных приложений. По сути архитектура нашего мастера показывает, как реализованы другие мастера Visual Studio .NET: ATL Simple Object Wizard (создание простых COM-объектов), Generic C++ Class Wizard (создание класса C++) и Add Member Variable Wizard (добавление переменной-члена).

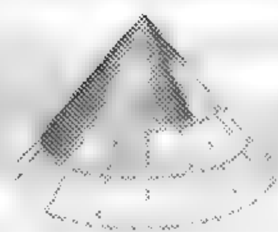
Всем этим мастерам доступны все объектные модели Visual Studio — именно через эту призму среда воспринимает классы и другой код вашего приложения.

Хорошенько «покопайтесь» в каталоге \Program Files\Microsoft Visual Studio .NET\VC7\VCWizards — там вы найдете все мастера Visual Studio .NET.

ГЛАВА

5





Сопоставление сообщений Windows

В главе 3 вы узнали, как каркас приложений MFC-библиотеки вызывает виртуальную функцию *OnDraw* класса «вид». Заглянув в интерактивную справочную систему по библиотеке MFC, вы узнаете, что класс *CView* и его базовый класс *CWnd* содержат несколько сотен функций-членов. Функции, имена которых начинаются с *On* — скажем, *OnKeyDown* и *OnLButtonUp*, — вызывает каркас приложения в ответ на события Windows вроде нажатий клавиш и щелчков мышью.

По большей части это не виртуальные функции, и поэтому они требуют дополнительных усилий при программировании. В данной главе мы покажем, как в окне Properties утилиты Class View создать структуру *карты сообщений* (message map) для подключения кода ваших функций к каркасу приложения.

В первых двух примерах этой главы используется обычный класс *CView*. В примере Ex05a мы обсудим взаимодействие инициируемых конечным пользователем событий и функций *OnDraw*. Пример Ex05b познакомит вас с результатом применения различных режимов *преобразования координат* (mapping modes) в Windows.

В большинстве практических задач вам понадобятся *окна с прокруткой* (scrolling view). Поэтому в примере Ex05c вместо *CView* используется класс *CScrollView*, позволяющий каркасу приложения библиотеки MFC разместить в окне линейки прокрутки и связать их с изображением.

Прием вводимых пользователем данных: функции карты сообщений

Приложение Ex03a из главы 3 не реагирует на действия конечного пользователя (за исключением стандартных команд Windows для изменения размеров и закрытия окна). Окно содержит меню и панель инструментов, но они не «подключены» к

коду класса «вид». Меню и панели инструментов мы изучим лишь в третьей части, так как они связаны с классом «рамка», но в Windows много других источников входной информации, которые не дадут вам расслабиться. Однако прежде чем вы сможете обработать какое-либо событие Windows, хотя бы и щелчок, вам нужно научиться пользоваться системой карт сообщений MFC.

Карта сообщений

Когда пользователь нажимает левую кнопку мыши в окне представления, Windows посылает этому окну сообщение, а именно *WM_LBUTTONDOWN*. Если в ответ на это сообщение программа должна выполнить какое-то действие, то в классе «вид» надо предусмотреть функцию:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // код обработки сообщения
}
```

а в заголовочном файле класса — указать соответствующий прототип:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

Элемент *afx_msg* преобразуется препроцессором в пустую строку и напоминает о том, что перед нами функция карты сообщений. Далее в коде программы должен присутствовать макрос карты сообщений, подключающий функцию *OnLButtonDown* к каркасу приложения:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_LBUTTONDOWN() // Запись для OnLButtonDown
    // Другие записи карты сообщений
END_MESSAGE_MAP()
```

И, наконец, заголовочный файл класса должен содержать оператор:

```
DECLARE_MESSAGE_MAP()
```

Как же узнать, какая именно функция соответствует определенному сообщению Windows? В приложении А (а также в интерактивной справочной системе библиотеки MFC) вы найдете таблицу, где перечислены все стандартные сообщения Windows и прототипы соответствующих им функций-членов. Вы можете программировать функции обработки сообщений вручную — для некоторых сообщений это даже необходимо. К счастью, в Visual C++ .NET окно Properties инструмента Class View автоматизирует кодирование большинства функций карты сообщений.

Сохранение состояния объекта «вид»: переменные-члены класса

Если программа принимает вводимые пользователем данные, имеет смысл реализовать некоторую «визуальную» обратную связь. Функция *OnDraw* класса «вид» рисует изображение на основании текущего состояния объекта «вид», и это состояние может изменяться в результате действий пользователя. В реальном MFC-приложении состояние приложения обычно хранится в объекте-документе, но нам

еще далеко до этого. Пока же мы используем две переменные-члены класса «вид»: *m_rectEllipse* и *m_nColor*. Первая — это объект класса *CRect*, который содержит текущий ограничивающий прямоугольник для эллипса, вторая — целое число, которое задает текущий цвет эллипса.

Примечание По соглашению имена нестатических переменных-членов класса в библиотеке MFC начинаются с *m_*.

Наша функция карты сообщений будет переключать цвет эллипса (состояние объекта «вид») между серым и белым по щелчку левой кнопки мыши. Начальные значения *m_rectEllipse* и *m_nColor* задаются конструктором класса, а функция-член *OnLButtonDown* переключает цвет.

Примечание Почему для хранения состояния объекта «вид» не использовать глобальную переменную? Это может вызвать проблемы, если в приложении несколько таких объектов. Кроме того, инкапсуляция данных внутри объекта — одна из основ объектно-ориентированного программирования.

Инициализация переменных-членов класса «вид»

Лучшее место для инициализации переменной-члена класса — конструктор:

```
CMyView::CMyView() : m_rectEllipse(0, 0, 200, 200) {...}
```

Аналогично можно инициализировать *m_nColor*. Так как это переменная встроенного типа (целое), компилятор сгенерирует такой же код, как если задействовать в теле конструктора оператор присваивания.

Теория недействительного прямоугольника

Функция *OnLButtonDown* может сколько угодно переключать значение *m_nColor*, но если это все, что она делает, функция *OnDraw* не будет вызвана (при условии, конечно, что пользователь, к примеру, не изменит размеры окна). Функция *OnLButtonDown* должна вызывать функцию *InvalidateRect* (функция-член, наследуемая классом «вид» от *CWnd*). *InvalidateRect* инициирует отправку Windows-сообщения *WM_PAINT*, которое в классе *CView* сопоставлено вызову виртуальной функции *OnDraw*, которой при необходимости доступен параметр «недействительный прямоугольник», который был передан в *InvalidateRect*.

В Windows существует два способа оптимизации операций рисования. Во-первых, Windows обновляет только пикселы, расположенные внутри недействительного прямоугольника. Таким образом, чем меньше размеры этого прямоугольника (определяемые, скажем, обработчиком *OnLButtonDown*), тем быстрее он перерисовывается. Во-вторых, исполнение команд на рисование за пределами недействительного прямоугольника — напрасная трата времени. Чтобы получить недействительный прямоугольник, функция *OnDraw* может вызвать функцию-член *GetClipBox* класса *CDC* и таким образом избежать рисования объектов за пределами этого прямоугольника. Вспомните: *OnDraw* вызывается не только в ответ на вызов *InvalidateRect*, но и когда пользователь изменяет размеры окна или открывает

невидимые ранее его части. Итак, функция *OnDraw* отвечает за все рисование в окне и обязана обрабатывать любые передаваемые ей недействительные прямоугольники.

Для тех, кто программирует в Win32

Библиотека MFC позволяет легко связать ваши *переменные состояния* (state variables) с окном с помощью переменных-членов C++. В Win32 для этого обычно применяют элементы *cbClsExtra* и *cbWndExtra* структуры *WNDCLASS*, но код, реализующий этот механизм, столь сложен, что разработчики, как правило, прибегают к глобальным переменным.

Клиентская область окна

Клиентская область (client area) — прямоугольная часть окна, в которую не входят рамка, заголовок, меню и стыкуемые панели инструментов. Ее размеры задает функция-член *GetClientRect* класса *CWnd*. Обычно рисовать за пределами этой области не разрешается, да и большинство сообщений мыши поступает в окно, только когда ее указатель находится в пределах этого окна.

Арифметические операции с *CRect*, *CPoint* и *CSize*

Классы *CRect, CPoint* и *CSize* — производные от структур Windows *RECT, POINT* и *SIZE* и поэтому наследуют такие целочисленные переменные-члены:

CRect	*left, top, right, bottom*
CPoint	*x, y*
CSize	*cx, cy*

В справочнике *Microsoft Foundation Class Reference* для этих классов определено множество перегруженных операторов. Вы можете, в частности:

- прибавлять объект *CSize* к объекту *CPoint*;
- вычитать объект *CSize* из объекта *CPoint*;
- вычитать один объект *CPoint* из другого, в результате получая объект *CSize*;
- прибавлять к объекту *CRect* объект *CPoint* или *CSize*;
- вычитать из объекта *CRect* объект *CPoint* или *CSize*.

Класс *CRect* содержит связанные с классами *CPoint* и *CSize* функции-члены. Например, функция-член *TopLeft* возвращает объект *CPoint*, функция *Size* — объект *CSize*. Таким образом, становится понятно, что объект *CSize* — это «разница» между двумя объектами *CPoint*, а объект *CRect* можно «сместить» на *CPoint*.

Попадает ли точка внутрь прямоугольника?

В классе *CRect* есть функция *PtInRect*, проверяющая, попадает ли точка в прямоугольник. Второй параметр *OnLButtonDown* — *point* — это объект класса *CPoint*, задающий место указателя мыши в клиентской области окна. Чтобы узнать, находится ли точка внутри прямоугольника *m_rectEllipse*, нужно сделать так:

```
if (m_rectEllipse.PtInRect(point)) {
   //Точка расположена внутри прямоугольника
}
```

Однако, как вы скоро поймете, этой простой проверки хватает, только если вы работаете в координатах устройства (что пока верно).

Оператор *CRect LPCRECT*

В справочнике *Microsoft Foundation Class Library Reference* отмечено, что *CWnd::InvalidateRect* принимает параметр *LPCRECT* (указатель на структуру *RECT*), а не *CRect*. Но *CRect* допускается как параметр, так как в классе *CRect* определен перегруженный оператор *LPCRECT()*, возвращающий адрес объекта *CRect*, что эквивалентно адресу объекта *RECT*. Поэтому компилятор, если надо, автоматически преобразует аргументы *CRect* в *LPCRECT*, и функции можно вызывать так, как если б они имели в качестве параметров ссылки на *CRect*.

Следующий фрагмент кода функции-члена класса «вид» получает координаты клиентского прямоугольника и сохраняет их в *rectClient*:

```
CRect rectClient;
GetClientRect(rectClient);
```

Попадает ли точка внутрь эллипса?

Код примера Ex05a определяет, нажата ли кнопка мыши внутри прямоугольника. Корректнее было бы проверить, попал ли указатель мыши в эллипс. Для этого нужно создать объект класса *CRgn*, соответствующий эллипсу, и затем вместо *PtInRect* вызвать функцию *PtInRegion*. Вот этот фрагмент программы:

```
CRgn rgn;
rgn.CreateEllipticRgnIndirect(m_rectEllipse);
if (rgn.PtInRegion(point)) {
   //Точка расположена внутри эллипса
}
```

CreateEllipticRgnIndirect — еще одна функция, принимающая параметр *LPCRECT*. Она создает специальную внутреннюю структуру *области* (region) Windows — эллиптическую область внутри окна. Затем эта структура связывается с объектом C++ *CRgn* в программе. (Структуру этого же типа применяют и для представления многоугольника.)

Пример Ex05a

В примере Ex05a эллипс (здесь он оказывается кругом) изменяет цвет по щелчку левой кнопки мыши, а указатель мыши находится в прямоугольнике, описанном вокруг эллипса. Для хранения состояния служат переменные-члены класса «вид», а для перерисовки изображения — функция *InvalidateRect*.

В примере Ex03a в главе 3 рисование в окне зависело только от одной функции *OnDraw*. В примере Ex05a нам понадобятся три функции (в том числе конструктор) и две переменные-члены. Полный текст файлов заголовка и реализации

для класса *CEx05aView* показан ниже. Все изменения по сравнению с кодом, сгенерированным MFC Application Wizard, а также *OnLButtonDown*, выделены.

Ex05aView.H

```
// Ex05aView.h : interface of the CEx05aView class
//

#pragma once

class CEx05aView : public CView
{
protected: // create from serialization only
    CEx05aView();
    DECLARE_DYNCREATE(CEx05aView)

// Attributes
public:
    CEx05aDoc* GetDocument() const;


// Operations
public:

// Overrides
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// Implementation
public:
    virtual ~CEx05aView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    DECLARE_MESSAGE_MAP()
private:
    int m_nColor;
    CRect m_rectEllipse;
```

```
[illegible]
[illegible]
   [illegible]
[illegible]

;
```

Ex05aView.cpp

```
[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
   [illegible]
[illegible]
[illegible]
[illegible]

[illegible] m_rectEllipse(0, 0, 200, 200)
{
   m_nColor = GRAY_BRUSH;
}

[illegible]
{
}
[illegible]
{
   [illegible]
```

см. след. стр.

```
    [illegible]

    [illegible]
}

//////////////////////////////////////////////////////////////////////
[illegible]

[illegible] pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}

//////////////////////////////////////////////////////////////////////
[illegible]

[illegible]
{
    [illegible]
    [illegible]
}

[illegible]
{
    [illegible]
}

[illegible]
{
    [illegible]
}

//////////////////////////////////////////////////////////////////////
[illegible]

[illegible]
[illegible]
{
    [illegible]
}

[illegible]
{
    [illegible]
}
[illegible]
{
    [illegible]
    [illegible]
```

```
}
#endif //_DEBUG

///////////////////////////////////////////////////////////////////////
// CEx05aView message handlers

void CEx05aView::OnLButtonDown(UINT nFlags, CPoint point)
{
   if (m_rectEllipse.PtInRect(point)) {
      if (m_nColor == GRAY_BRUSH) {
         m_nColor = WHITE_BRUSH;
      }
      else {
         m_nColor = GRAY_BRUSH;
      }
      InvalidateRect(m_rectEllipse);
   }
}
```

Использование Class View с Ex05a

Взгляните на фрагмент Ex05aView.h:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

и на фрагмент Ex05aView.cpp:

```
ON_WM_LBUTTONDOWN()
```

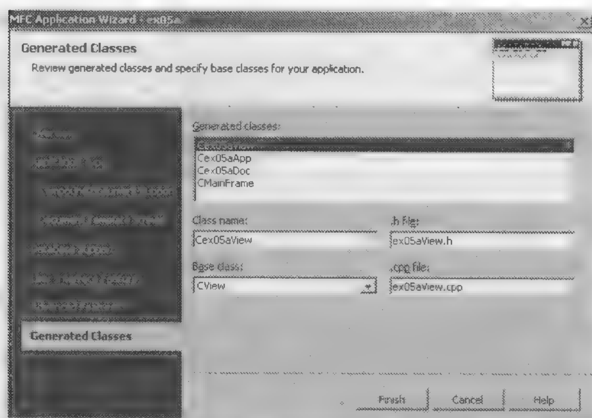
В предыдущей версии Visual C++ мастер AppWizard размещал здесь специальные комментарии для ClassWizard. К счастью, потребность в подобных комментариях отпала: Visual C++ .NET постоянно отслеживает состояние всего кода, в том числе сопоставленния функций и отдельных строк исходного текста. Мастера исходного кода, доступные в окне Properties средства Class View, добавляют прототипы обработчиков на основании внутренней информации. Кроме того, они генерируют шаблон функции-члена *OnLButtonDown* в Ex05aView.cpp, который содержит соответствующие объявления типов параметров и возвращаемого значения.

Обратите внимание на отличие связки между MFC Application Wizard и мастерами исходного текста от обычного генератора исходных текстов. Обычный генератор запускается один раз, после чего программист редактирует полученный код. MFC Application Wizard запускается для генерации приложения только раз, но мастера Class View можно использовать сколько угодно, и отредактировать код можно в любой момент.

Совместное использование MFC Application Wizard и мастеров исходного текста

Ниже описана последовательность создания приложения Ex05a с помощью MFC Application Wizard и мастеров, доступных в окне Properties средства Class View.

1. **Создайте Ex05a, используя MFC Application Wizard.** Сгенерируйте с помощью MFC Application Wizard SDI-проект с именем Ex05a в подкаталоге \vcpp32\ex05a. Параметры и имена классов по умолчанию показаны ниже.

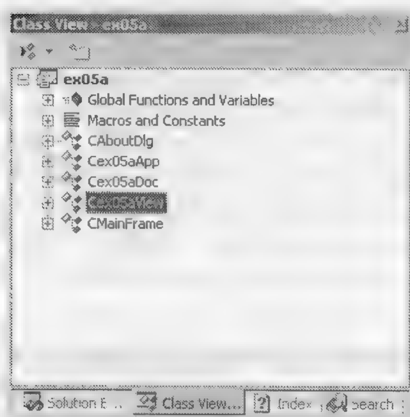


2. **Добавьте к *CEx05aView* переменные-члены *m_rectEllipse* и *m_nColor*.** В меню View выберите Class View, щелкните правой кнопкой класс *CEx05aView*, в контекстном меню выберите Add Variable и вставьте две переменные-члены:

```
private:
    CRect m_rectEllipse;
    int m_nColor;
```

   Впрочем, этот код в объявление класса в Ex05aView.h можно ввести вручную.

3. **В окне Propeties средства Class View добавьте обработчик сообщения в класс *CEx05aView*.** В Class View выберите класс *CEx05aView*, как показано на рисунке, щелкните его правой кнопкой и в контекстном меню выберите Properties. Щелкните кнопку Messages на инструментальной панели Properties. Выберите в списке запись *WM_LBUTTONDOWN*. Рядом ней появится стрелочка поля со списком — выберите в нем <Add> OnLButtonDown. В исходный код добавится функция *OnLButtonDown*, текст которой появится в окне редактора исходного кода Code Editor.



4. **Отредактируйте код функции *OnLButtonDown* в файле Ex05aView.cpp.** В открывшемся окне Code Editor замените код функции на выделенный (введите его вручную):

```
void CEx05aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (m_rectEllipse.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(m_rectEllipse);
    }
}
```

5. **Отредактируйте конструктор и функцию *OnDraw* в файле Ex05aView.cpp.** Вместо сгенерированного кода надо вручную ввести выделенный код:

```
CEx05aView::CEx05aView() : m_rectEllipse(0, 0, 200, 200)
{
    m_nColor = GRAY_BRUSH;
}
:
void CEx05aView::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}
```

Для тех, кто программирует в Win32

Обычное приложение для Windows регистрирует набор *оконных классов* (windows classes) (не путать с классами C++!) и назначает при этом каждому классу уникальную функцию — *оконную процедуру* (window procedure). Всякий раз, вызывая *CreateWindow* для создания окна, приложение указывает в качестве параметра оконный класс, связывая таким образом вновь созданное окно с оконной процедурой. Эта функция, вызываемая всякий раз, когда Windows посылает окну сообщение, проверяет код сообщения, передаваемый ей как параметр, и выполняет соответствующую обработку сообщения.

В каркасе приложения MFC для большинства типов окон используется один оконный класс и одна оконная процедура. Эта процедура отыскивает описатель окна (передаваемый как параметр) в *карте описателей MFC* (MFC handle map) для получения указателя на соответствующий объект-окно C++. Затем оконная процедура использует MFC-систему *классов периода исполнения* (runtime class), чтобы определить класс C++ объекта-окна. Далее она отыскивает функцию-обработчик в статических картах сообщений и вызывает эту функцию для соответствующего объекта-окна.

6. **Соберите и запустите программу.** В меню Build выберите команду Build Ex05a или щелкните на панели инструментов Build кнопку:



Выберите в меню Debug команду Start Without Debugging. Полученная программа в ответ на щелчок изменяет цвет круга в окне представления. (Не нажимайте кнопку два раза подряд слишком быстро — Windows интерпретирует это как один двойной щелчок, а не как два одиночных.)

Режимы преобразования координат

До этого момента мы задавали координаты для рисования в пикселах экрана или в так называемых *координатах устройства* (device coordinates). Ex05a использует в качестве координат пикселы, так как в контексте устройства установлен *режим преобразования координат* (mapping mode) по умолчанию — *MM_TEXT*. Следующий оператор рисует квадрат 200×200 пикселов, левый верхний угол которого совпадает с левым верхним углом клиентской области окна (ось ординат *y* направлена вниз):

```
pDC->Rectangle(CRect(0, 0, 200, 200));
```

На дисплее с разрешением 1024×768 пикселов такой квадрат будет выглядеть меньше, чем на стандартном мониторе VGA с разрешением 640×480, а при распечатке на лазерном принтере с разрешением 600 dpi он покажется вовсе крошечным. [Убедитесь в этом сами, выбрав команду программы Ex05a — функцию Print Preview (предварительный просмотр перед печатью)].

А если размер квадрата должен быть 4×4 см независимо от устройства отображения? Windows предоставляет ряд других режимов преобразования координат (или систем координат), которые выбирают для контекста устройства. Координаты в текущем режиме преобразования называются *логическими координатами* (logical coordinates). Если, например, выбрать режим преобразования *MM_HIMETRIC*, логической единицей станет не пиксел, а 0,01 мм. В режиме *MM_HIMETRIC* ось *y* направлена в противоположную сторону по сравнению с режимом *MM_TEXT*: при перемещении вниз значения *y* уменьшаются. Так что в логических координатах квадрат 4×4 см можно нарисовать так:

```
pDC->Rectangle(CRect(0, 0, 4000, -4000));
```

Выглядит просто, да? На самом деле это не так, потому что работать только в логических координатах нельзя. Программа постоянно переключается между координатами устройства и логическими координатами, и вы должны знать, когда выполнить соответствующее преобразование. Далее мы рассмотрим несколько правил, которые облегчат вашу нелегкую долю программиста. Прежде всего познакомимся с тем, какие же режимы преобразования координат есть в Windows.

Режим преобразования *MM_TEXT*

На первый взгляд, *MM_TEXT* — это вовсе и не режим преобразования координат, а лишь другое название для аппаратной системы координат. Верно, но не совсем. В режиме *MM_TEXT* координаты соответствуют пикселам, значения по оси *x* воз-

растают при движении вправо, значения по оси *y* возрастают при движении вниз, но можно переместить начало координат, используя функции *SetViewportOrg* и *SetWindowOrg* класса *CDC*. Ниже приведен пример программы, устанавливающей начало логических координат в точку (100, 100) и рисующей затем квадрат с размерами 200x200 пикселов со смещением (100, 100) (рис. 5-1). Логическая точка (100, 100) соответствует точке (0, 0) в аппаратных координатах. Подобное преобразование применяется в окне с прокруткой.

```
void CMyView::OnDraw(CDC* pDC)
{
    pDC->SetMapMode(MM_TEXT);
    pDC->SetWindowOrg(CPoint(100, 100));
    pDC->Rectangle(CRect(100, 100, 300, 300));
}
```

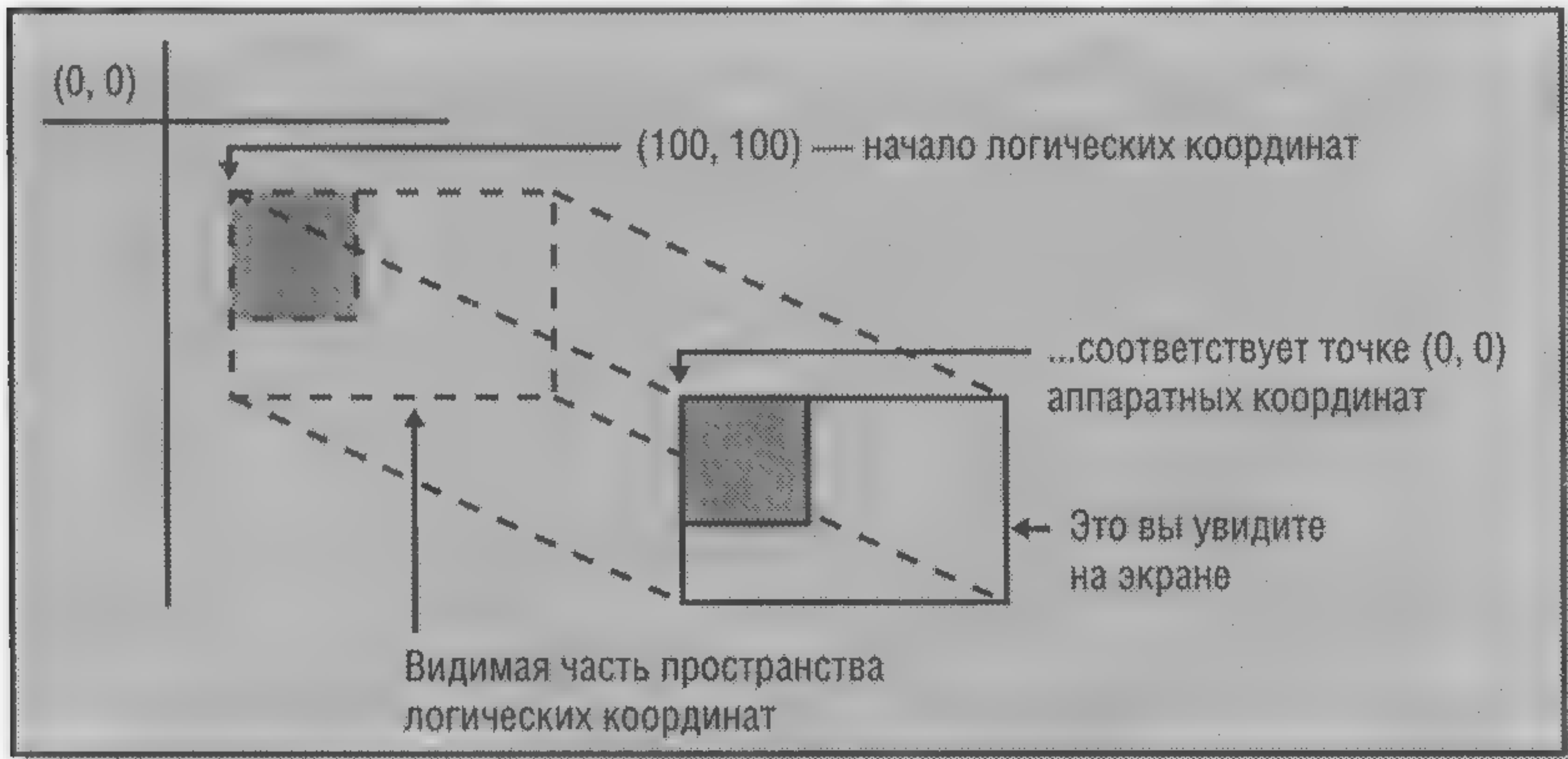


Рис. 5-1. *Рисование квадрата после того, как начало координат перемещено в точку (100, 100)*

Режимы преобразования координат с постоянным масштабом

Важная группа режимов преобразования координат Windows — режимы с постоянным масштабом. Как вы уже видели, в режиме *MM_HIMETRIC* значения *x* возрастают при перемещении вправо, а значения *y* убывают при перемещении вниз. Это соглашение соблюдается во всех режимах с постоянным масштабом, и изменить его нельзя. Единственное различие между режимами с постоянным масштабом — фактический масштабный множитель, который определяется, как показано в табл. 5-1.

Табл. 5-1. Масштабный множитель различных режимов преобразования координат

Режим преобразования координат	Логическая единица
MM_LOENGLISH	0,01 дюйма
MM_HIENGLISH	0,001 дюйма
MM_LOMETRIC	0,1 мм
MM_HIMETRIC	0,01 мм
MM_TWIPS	1/1440 дюйма

Последний режим преобразования — *MM_TWIPS* — чаще всего используется при работе с принтерами. Один twip равен 1/20 пункта. [*Пункт* (point) (сокращенно *пт*), — единица измерения размера шрифтов. В Windows 1 пт = 1/72 дюйма.] Если в режиме *MM_TWIPS* нужен шрифт размером 12 пт, установите высоту символа в 12×20, т. е. 240 twips.

Режимы преобразования координат с переменным масштабом

Windows предоставляет два режима преобразования координат, позволяющих изменять не только начало координат, но и масштабный множитель: *MM_ISOTROPIC* и *MM_ANISOTROPIC*. В этих режимах можно изменять размеры рисунка, когда пользователь изменяет размеры окна, а также переворачивать изображение, изменяя знак масштабного множителя или задавать произвольные масштабные множители.

В режиме *MM_ISOTROPIC* всегда поддерживается коэффициент пропорциональности между осями, равный 1:1. Иначе говоря, при изменении масштабного множителя круг все равно остается кругом. В режиме *MM_ANISOTROPIC* масштабные множители по осям x и y изменяются независимо друг от друга. Круги могут вырождаться в эллипсы.

Эта функция *OnDraw* рисует эллипс, точно вписывающийся в размеры окна:

```
void CMyView::OnDraw(CDC* pDC)
{
    CRect rectClient;
    GetClientRect(rectClient);
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1000, 1000);
    pDC->SetViewportExt(rectClient.right, -rectClient.bottom);
    pDC->SetVieportOrg(rectClient.right / 2, rectClient.bottom / 2);

    pDC->Ellipse(CRect(-500, -500, 500, 500));
}
```

Как она работает? Функции *SetWindowExt* и *SetViewportExt* совместными усилиями задают масштабные множители в зависимости от текущего размера клиентской области окна, который возвращает функция *GetClientRect*. В результате размер окна равен точно 1000×1000 логических единиц. Функция *SetViewportOrg* устанавливает начало координат в центр окна. Так что эллипс с радиусом 500 логических единиц и с центром в центре окна целиком заполняет окно (рис. 5-2).

Вот формулы преобразования логических координат в координаты устройства:

- <масштаб по x> = <размер области вывода по x> / <размер окна по x>;
- <масштаб по y> = <размер области вывода по y> / <размер окна по y>;
- <аппаратная координата по x> = <логическая координата по x> * <масштаб по x> + <смещение начала координат по x>;
- <аппаратная координата по y> = <логическая координата по y> * <масштаб по y> + <смещение начала координат по y>.

Допустим, ширина окна 448 пикселов (*rectClient.right*). Правый край клиентской области эллипса находится в 500 логических единицах от начала координат. Масштаб по оси x равен 448/1000, а смещение начала координат по x — 448/2 ап-

паратных единиц. По приведенным формулам координата правого края клиентской области эллипса окажется равной 448 аппаратным единицам, т. е. координате правого края окна. Масштаб по x выражен как отношение «размер области вывода/размер окна», так как координаты в Windows — это целые числа, а не величины с плавающей запятой. Сами по себе размеры области вывода или окна смысла не имеют.

Если в предыдущем примере заменить *MM_ANISOTROPIC* на *MM_ISOTROPIC*, то «эллипс» всегда будет кругом (рис. 5-3), а его радиус — равным длине меньшего измерения окна.

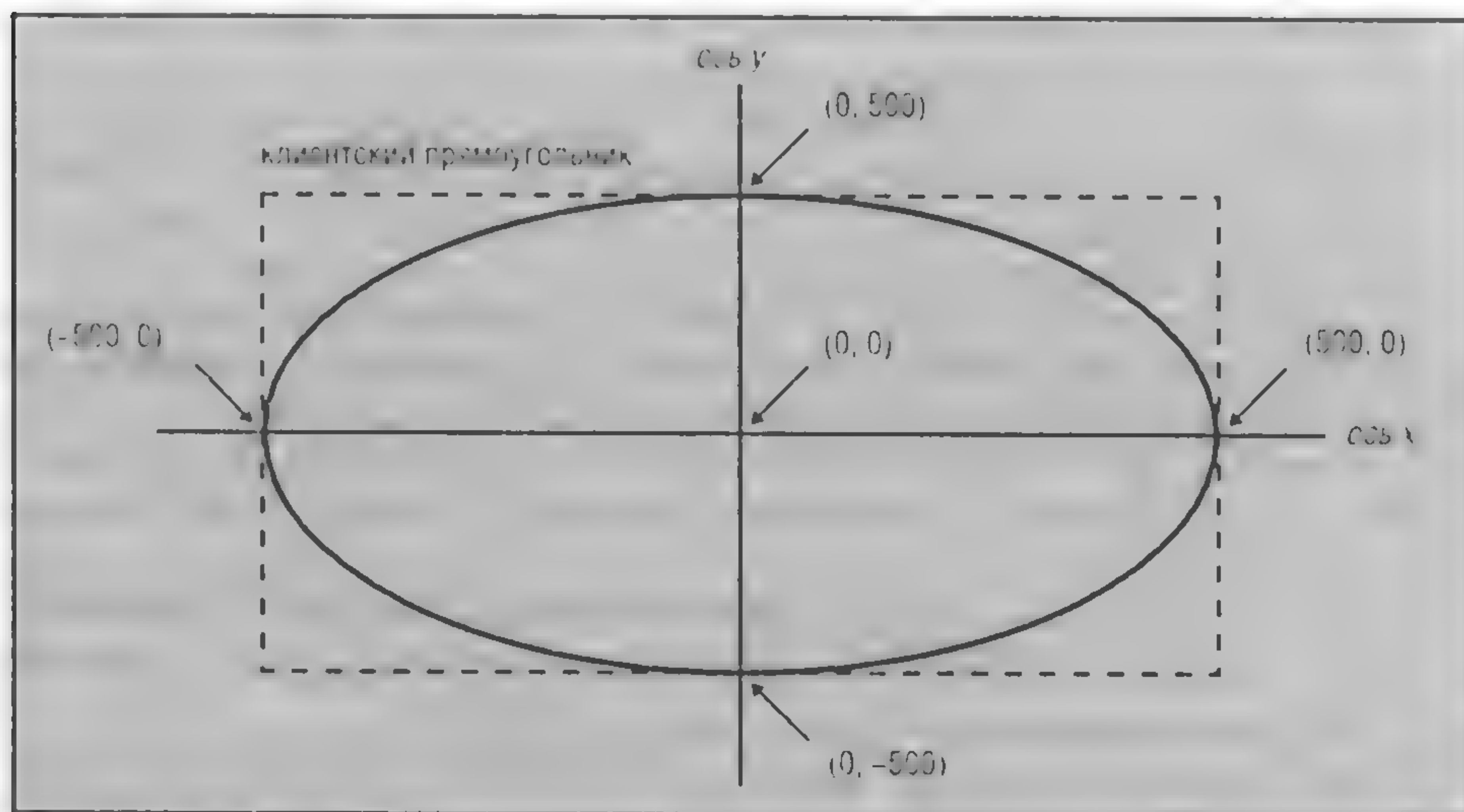


Рис. 5-2. *Центрированный эллипс в режиме преобразования координат* MM_ANISOTROPIC

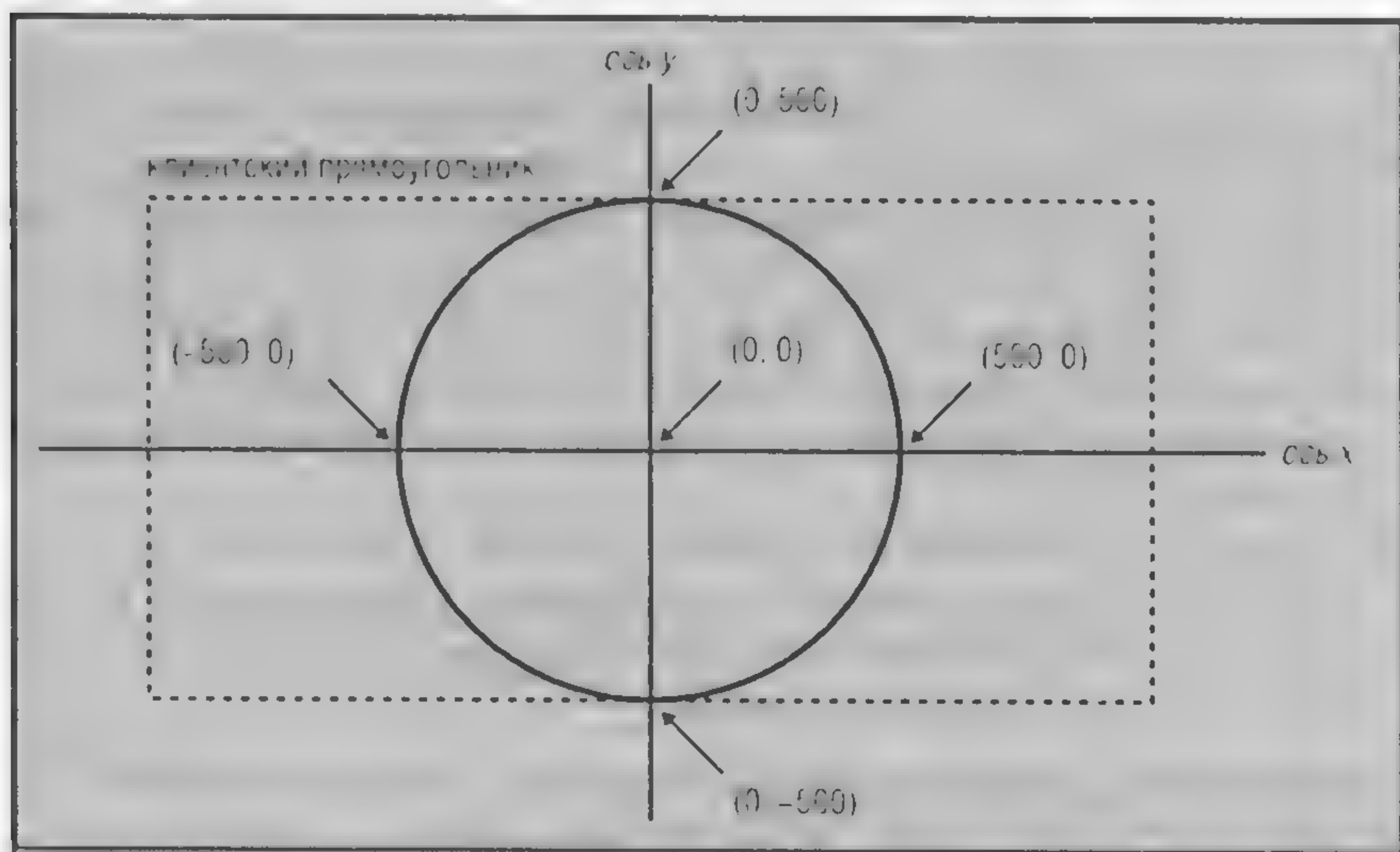


Рис. 5-3. *Центрированный эллипс, нарисованный в режиме преобразования координат* MM_ISOTROPIC

Преобразование координат

После определения режима преобразования координат (и начала координат) большинству функций-членов *CDC* можно в качестве параметров передавать логические координаты. Однако если вы получаете координаты курсора мыши из сообщения Windows (параметр *point* в *OnLButtonDown*), это аппаратные координаты. Корректная работа многих других функций MFC, в частности функций-членов класса *CRect*, возможна только в аппаратных координатах.

Примечание Win32-функции арифметики *CRect* используют соответствующие арифметические функции Win32 для *RECT*, в которых подразумевается, что *right* больше *left*, а *bottom* больше *top*. Скажем, для прямоугольника (0, 0, 1000, -1000) в координатах *MM_HIMETRIC* значение *bottom* меньше *top*, и этот прямоугольник не удастся обработать функциями вроде *CRect::PtInRect*, если предварительно не вызвать *CRect::NormalizeRect*, чтобы изменить значения переменных-членов *CRect* на (0, -1000, 1000, 0).

Более того, скорее всего понадобится третий набор координат — физических. Зачем? Допустим, вы используете режим *MM_LOENGLISH*, где логическая единица равна 0,01 дюйма, но 1 дюйм на экране представляет собой фут (12 дюймов) в реальном мире. Далее. Пусть пользователь работает с дюймами и десятичными дробями. Значение 26,75 дюймов преобразуется в 223 логические единицы, которые затем нужно преобразовать в координаты устройства. Во избежание ошибок округления физические координаты следует хранить либо как числа с плавающей запятой, либо как целые (long) с масштабным множителем.

Преобразования физических координат в логические остаются на вашей совести, а вот о преобразовании логических координат в аппаратные позаботится GDI Windows. Преобразование между двумя системами выполняют функции *LPtoDP* и *DPtoLP* класса *CDC*, при этом предполагается, что режим преобразования координат и связанные с этим параметры контекста уже заданы. Ваша задача — решить, когда какую из систем координат использовать. Вот несколько правил.

- Считайте, что все параметры, передаваемые в функции-члены *CDC*, — это логические координаты.
- Считайте, что все параметры, передаваемые в функции-члены *CWnd*, — это аппаратные координаты;
- Проверяя, попадает ли указатель мыши в определенную область, используйте аппаратные координаты. Задавайте области в аппаратных координатах. Такие функции, как *CRect::PtInRect*, лучше всего работают, если применяются аппаратные координаты.
- Значения, сохраняемые на длительное время, должны использовать логические или физические координаты. Если вы сохранили значение точки в аппаратных координатах и пользователь прокрутил изображение в окне, то сохраненное значение станет недействительным.

Пусть нам надо узнать, находится ли указатель мыши в момент нажатия левой кнопки в прямоугольнике. Вот соответствующий код.

```
// m_rect - это переменная-член типа CRect производного класса "вид",
// содержащая логические координаты MM_LOENGLISH

void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect = m_rect;          // rect - временная копия m_rect
    CClientDC dc(this);           // Получение контекста устройства
                                  // для SetMapMode и LPtoDP
                                  // - подробности в следующей главе
    dc.SetMapMode(MM_LOENGLISH);
    dc.LPtoDP(rect);              // Теперь rect содержит координаты устройства
    if (rect.PtInRect(point)) {
        TRACE("Mouse cursor is inside the rectangle.\n");
    }
}
```

Обратите внимание на использование макроса *TRACE* (описан в главе 2).

Примечание Как вы скоро увидите, режим преобразования координат лучше устанавливать не в функции *OnDraw*, а в виртуальной функции *OnPrepareDC* класса *CView*.

Пример Ex05b: переход в режим преобразования координат *MM_HIMETRIC*

Ex05b представляет собой пример Ex05a, модифицированный для поддержки преобразования координат в режиме *MM_HIMETRIC*. В проекте Ex05b на компакт-диске имена классов и файлов другие, однако ниже рассказано, как преобразовать код проекта Ex05a. Как и Ex05a, Ex05b выполняет проверку на попадание указателя мыши в заданную область и изменяет цвет эллипса только при щелчке внутри описанного прямоугольника.

1. **В окне Properties инструмента Class View переопределите виртуальную функцию *OnPrepareDC*.** Class View позволяет переопределять виртуальные функции некоторых базовых классов MFC, в том числе *CView*, в окне Properties. Соответствующий мастер генерирует прототип функции в заголовочном файле класса и шаблон тела функции в CPP-файле. В окне Class View щелкните правой кнопкой имя класса *CEx05aView* и в контекстном меню выберите Properties. Щелкнув кнопку Overrides на панели инструментов окна Properties, выберите в списке функцию *OnPrepareDC* и отредактируйте ее следующим образом:

```
void CEc05aView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
        pDC->SetMapMode(MM_HIMETRIC);
        CView::OnPrepareDC(pDC, pInfo);
}
```

   Каркас приложения вызывает виртуальную функцию *OnPrepareDC* прямо перед вызовом *OnDraw*.

2. **Отредактируйте конструктор класса «вид».** Надо изменить значения координат прямоугольника эллипса. Теперь размер прямоугольника равен 4×4 см, а не 200×200 пикселов. Заметьте: значение *y* должно быть отрицательным, иначе эллипс будет выведен на «виртуальный экран», расположенный непосредственно над монитором! Измените значения, как показано ниже:

```
CEx05aView::CEx05aView() : m_rectEllipse(0, 0, 4000, -4000)
{
      m_nColor = GRAY_BRUSH;
}
```

3. **Отредактируйте функцию *OnLButtonDown*.** Теперь для проверки попадания в прямоугольник она должна преобразовать координаты эллипса в координаты устройства. Измените функцию следующим образом:

```
void CEx05aView::OnLButtonDown(UINT nFlags, CPoint point)
{
      CClientDC dc(this);
      OnPrepareDC(&dc);
      CRect rectDevice = m_rectEllipse;
      dc.LPtoDP(rectDevice);
      if (rectDevice.PtInRect(point)) {
         if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
         }
         else {
            m_nColor = GRAY_BRUSH;
         }
         InvalidateRect(rectDevice);
      }
}
```

4. **Соберите и запустите программу Ex05b**. Программа работает так же, как и Ex05a, за исключением того, что размер эллипса другой. В режиме предварительного просмотра печати вы увидите, что эллипс гораздо больше, чем в Ex05a.

Окно представления с прокруткой

Как следует из отсутствия *линеек прокрутки* (scroll bars) в примерах Ex05a и Ex05b, MFC-класс *CView* — базовый класс для *CEx05bView* — сам прокрутку не поддерживает. Ее поддерживает другой класс — *CScrollView*. *CScrollView* — наследник *CView*. Мы создадим новую программу Ex05c, использовав *CScrollView* вместо *CView*. Код преобразования координат, написанный для Ex05b, подготовил все, что нужно для реализации прокрутки.

Класс *CScrollView* поддерживает прокрутку через линейки прокрутки, но не с клавиатуры. Добавить поддержку прокрутки с клавиатуры легко, и мы это сделаем.

Окно — это больше, чем видно на экране

Если мышью уменьшить размеры обычного окна, его содержимое останется неподвижным относительно его левого верхнего уровня, а элементы, расположен-

ные снизу и справа, исчезнут из поля зрения. После увеличения окна они появятся вновь. Отсюда логично заключить, что окно больше, чем его *область вывода* (viewport), которую вы видите на экране. Однако область вывода не обязана быть жестко привязанной к левому верхнему краю окна. Благодаря функциям *ScrollWindow* и *SetWindowOrg* объекта *CWnd*, класс *CScrollView* позволяет перемещать область вывода в любое место окна, включая области, расположенные выше и левее.

Линейки прокрутки

Microsoft Windows позволяет легко отобразить линейки прокрутки по краям окна, однако Windows сама по себе не пытается подключить их к окну. Эту задачу выполняет класс *CScrollView*. Функции-члены *CScrollView* обрабатывают сообщения *WM_HSCROLL* и *WM_VSCROLL*, которые линейки прокрутки посылают объекту «вид». Эти функции перемещают область вывода внутри окна и выполняют необходимые вспомогательные действия.

Различные способы прокрутки

Класс *CScrollView* поддерживает один определенный способ прокрутки, в котором используется одно большое окно и маленькая область вывода. Для каждого элемента определено положение в большом окне. Например, если нужно отобразить на экране 10 000 адресных строк, то вместо окна длиной в 10 000 строк, вероятно, лучше иметь небольшое окно, поддерживающее алгоритм прокрутки, который выбирает для отображения столько строк, сколько можно отобразить в данный момент. В нашем случае надо создать свой производный от *CView* класс «вид» с прокруткой.

Функция *OnInitialUpdate*

Подробнее о ней вы узнаете при изучении архитектуры «документ-вид», которое мы начнем с главы 15. Здесь виртуальная функция *OnInitialUpdate* важна потому, что к ней первой обращается каркас приложения по завершении создания окна представления, но перед вызовом *OnDraw*, так что именно в *OnInitialUpdate* следует задать логический размер и режим преобразования координат для вывода с прокруткой. Эти параметры устанавливает функция *CScrollView::SetScrollSizes*.

Прием данных, вводимых с клавиатуры

Прием данных, вводимых с клавиатуры, — двухэтапный процесс. Windows направляет в окно сообщения *WM_KEYDOWN* и *WM_KEYUP* с *кодами виртуальных клавиш* (virtual key codes), но на пути к окну эти сообщения преобразуются. Если введен символ ANSI (в результате чего генерируется сообщение *WM_KEYDOWN*), функция преобразования проверяет состояние регистра клавиатуры и направляет сообщение *WM_CHAR* с кодом соответствующего символа — либо верхнего, либо нижнего регистра. Клавиши перемещения курсора и функциональные клавиши не имеют соответствующих кодов символов, поэтому для них преобразования не требуется. Окно получает только сообщения *WM_KEYDOWN* и *WM_KEYUP*.

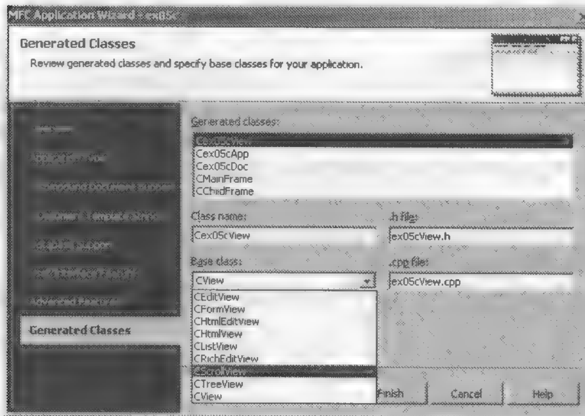
Создать в своем классе «вид» обработчики этих сообщений вы можете в окне Properties в Class View. Если предполагается принимать данные с алфавитно-циф-

ровых клавиш, обрабатывайте *WM_CHAR*; если нужно обрабатывать нажатия и других клавиш, обрабатывайте *WM_KEYDOWN*. Библиотека MFC предоставляет в качестве параметра функции-обработчика код символа или виртуальной клавиши.

Пример Ex05c: прокрутка

Задача Ex05c — создать логическое окно с размерами 20 см в ширину и 30 см в высоту. Программа рисует тот же эллипс, что и в Ex05b. Можно отредактировать исходные файлы Ex05b и заменить базовый класс *CView* на *CScrollView*, но проще начать снова с MFC Application Wizard, который и сгенерирует функцию, переопределяющую *OnInitialUpdate*.

1. **С помощью MFC Application Wizard создайте Ex05c.** Создайте SDI-проект Ex05c в подкаталоге \vcpp32\ex05c. Установите класс *CScrollView* в качестве базового для класса *CEx05View*:



2. **Добавьте в Ex05cView.h *переменные-члены m_rectEllipse* и *m_nColor*.** Вставьте следующий код средствами Add Member Variable Wizard в окне Properties или введите текст вручную в объявлении класса *CEx05cView*:

```
private:
    CRect m_rectEllipse;
    int m_nColor;
```

   Это те же переменные-члены, что мы добавляли в проектах Ex05a и Ex05b.

3. **Измените сгенерированную MFC Application Wizard функцию *OnInitialUpdate*.** Отредактируйте *OnInitialUpdate* в файле Ex05cView.cpp:

```
void CEx05cView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(20000, 30000); // 20 на 30 см
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizePage, sizeLine);
}
```

4. **В окне Properties в Class View добавьте обработчик сообщения *WM_KEYDOWN*.** Мастер создаст функцию-член *OnKeyDown*, а также соответствующий прототип и запись в карте сообщений. Отредактируйте код так:

```
void CEx05cView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
   switch (nChar) {
   case VK_HOME:
      OnVScroll(SB_TOP, 0, NULL);
      OnHScroll(SB_LEFT, 0, NULL);
      break;
   case VK_END:
      OnVScroll(SB_BOTTOM, 0, NULL);
      OnHScroll(SB_RIGHT, 0, NULL);
      break;
   case VK_UP:
      OnVScroll(SB_LINEUP, 0, NULL);
      break;
   case VK_DOWN:
      OnVScroll(SB_LINEDOWN, 0, NULL);
      break;
   case VK_PRIOR:
      OnVScroll(SB_PAGEUP, 0, NULL);
      break;
   case VK_NEXT:
      OnVScroll(SB_PAGEDOWN, 0, NULL);
      break;
   case VK_LEFT:
      OnHScroll(SB_LINELEFT, 0, NULL);
      break;
   case VK_RIGHT:
      OnHScroll(SB_LINERIGHT, 0, NULL);
      break;
   default:
      break;
   }
}
```

5. **Отредактируйте конструктор и функцию *OnDraw*.** Измените в файле Ex05cView.cpp сгенерированные MFC Application Wizard конструктор и функцию *OnDraw*:

```
CEx05cView::CEx05cView() : m_rectEllipse(0, 0, 4000, -4000)
{
   m_nColor = GRAY_BRUSH;
}
:
void CEx05cView::OnDraw(CDC* pDC)
{
   pDC->SelectStockObject(m_nColor);
   pDC->Ellipse(m_rectEllipse);
}
```

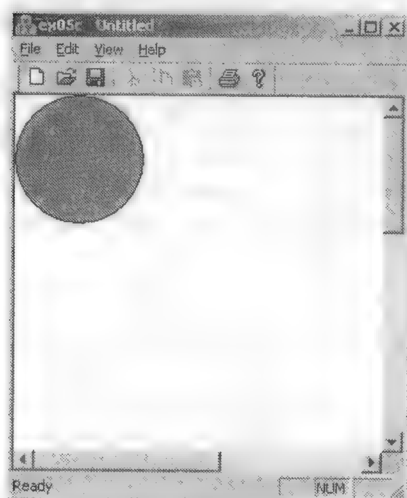
Эти функции идентичны аналогичным функциям из Ex05a и Ex05b.

6. **Создайте обработчик сообщения *WM_LBUTTONDOWN*.** Измените сгенерированный код:

```
void CEx05cView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);
    OnPrepareDC(&dc);
    CRect rectDevice = m_rectEllipse;
    dc.LPtoDP(rectDevice);
    if (rectDevice.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(rectDevice);
    }
}
```

Функция идентична обработчику *OnLButtonDown* из проекта Ex05b. Как и там, она вызывает *OnPrepareDC*, но здесь есть отличие. В классе *CEx05cView* нет переопределенной функции *OnPrepareDC*, поэтому здесь вызывается *CScrollView::OnPrepareDC*. В соответствии с первым параметром *SetScrollSizes* эта функция устанавливает режим преобразования координат, а также начало координат окна согласно текущей позиции прокрутки. Преобразование координат необходимо для коррекции смещения начала координат, даже если используется режим *MM_TEXT*.

7. **Соберите и запустите программу Ex05c.** Убедитесь, что щелчок мышью работает, даже когда в результате прокрутки круг выходит за пределы окна. Проверьте прокрутку с клавиатуры. Результат работы программы должен выглядеть, как показано на рисунке.



Другие сообщения Windows

Библиотека MFC напрямую поддерживает сотни функций обработки сообщений Windows. Кроме того, вы вправе определять собственные сообщения. В следующих главах вы найдете примеры обработки сообщений, в том числе сообщений от меню, дочерних окон-элементов управления и т. п. Пока же стоит обратить особое внимание на пять сообщений Windows: *WM_CREATE*, *WM_CLOSE*, *WM_QUERYENDSESSION*, *WM_DESTROY* и *WM_NCDESTROY*.

Сообщение *WM_CREATE*

Это первое сообщение, которое Windows посылает окну[1]. Это происходит, когда каркас приложения вызывает функцию *Create* окна, так что к этому моменту создание окна еще не завершено и окно невидимо. Следовательно, ваш обработчик *OnCreate* не может вызывать функции Windows, которые требуют уже готового окна.[2] Эти функции можно вызывать в переопределенной функции *OnInitialUpdate*, однако в SDI-приложениях последняя за время существования окна представления может вызываться неоднократно.

Сообщение *WM_CLOSE*

Windows посылает сообщение *WM_CLOSE*, когда пользователь закрывает окно через системное меню или когда закрывается родительское окно. Реализовав обработчик *OnClose* в производном классе, вы можете управлять процессом закрытия окна. Так, если нужно спросить у пользователя, сохранить ли изменения в файле, делайте это в *OnClose*. Только убедившись, что окно можно безопасно закрывать, вызывайте функцию *OnClose* базового класса, которая продолжит процесс закрытия. Объект «вид» и соответствующее ему окно в данный момент по-прежнему активны.

Примечание При использовании всех средств каркаса приложения вы, вероятно, не станете применять обработчик события *WM_CLOSE*. Вместо этого можно переопределить виртуальную функцию *CDocument::SaveModified* как часть высокоструктурированной процедуры завершения программы, реализованной каркасом приложения.

Сообщение *WM_QUERYENDSESSION*

Сообщение *WM_QUERYENDSESSION* направляется во все исполняющиеся приложения, когда пользователь завершает работу с Windows. Сообщение обрабатывается функцией *OnQueryEndSession* карты сообщений. Если вы пишете обработчик для *WM_CLOSE*, напишите обработчик и для *WM_QUERYENDSESSION*.

---

[1] Это неверно: первым сообщением будет либо *WM_NCCREATE*, либо *WM_GETMINMAXINFO*. — *Прим. перев.*

[2] Тоже ошибочное утверждение. Окно как структура Windows полностью создано, а библиотека MFC уже присвоила его описатель переменной *m_hWnd* класса *CWnd*, хотя возврата из функции *Create* еще не было. — *Прим. перев.*

Сообщение *WM_DESTROY*

Это сообщение, посылаемое Windows после сообщения *WM_CLOSE*, обрабатывается функцией карты сообщений *OnDestroy*. Получив его, следует считать, что окно представления уже невидимо, но все еще активно, как и его дочерние окна. Обработчик этого сообщения выполняет очистку, которая требует существования окна Windows. Обязательно вызовите функцию *OnDestroy* базового класса. Функция *OnDestroy* в классе «вид» не может отменить процесс уничтожения окна. Для этого служит *OnClose*.

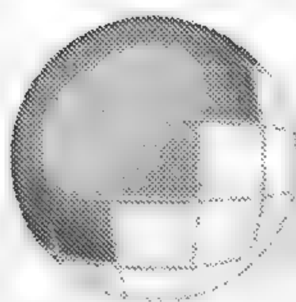
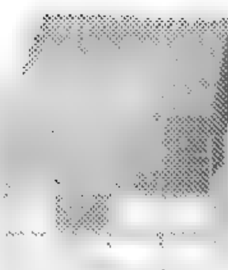
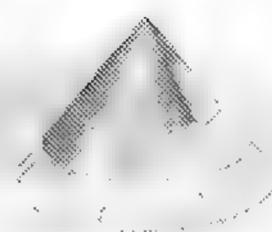
Сообщение WM_*NCDESTROY*

Это последнее сообщение, посылаемое Windows при уничтожении окна. Все дочерние окна уже уничтожены. В *OnNcDestroy* можно выполнить финальную обработку, которая не требует наличия активного окна. На забудьте вызвать функцию *OnNcDestroy* базового класса.

Примечание Не пытайтесь уничтожать в *OnNcDestroy* динамически созданный объект-окно. Это действие выполняет специальная виртуальная функция *PostNcDestroy* класса *CWnd*, вызываемая из базового класса *OnNcDestroy*. О том, когда удалять объект-окно, см. MFC Technical Note 17.

ГЛАВА

6

Классические функции графического устройства, шрифты и растровые изображения

Мы уже встречались с элементами интерфейса графического устройства (GDI). Всякий раз при выводе чего-нибудь на дисплей или на принтер программа использует функции GDI или GDI+. С функциями «классического» GDI мы познакомимся в этой главе, а с функциями GDI+ — в главе 33, когда начнем обсуждать .NET.

Эти функции позволяют рисовать точки, линии, прямоугольники, многоугольники, эллипсы, растровые изображения и вводить текст. Круги и квадраты вы сможете рисовать почти сразу, однако вывод текста — задача более сложная. Из этой главы вы узнаете, как эффективно использовать GDI в среде Microsoft Visual C++ .NET, работать со шрифтами на дисплее и на принтере.

Классы контекста устройства

В главах 3 и 4 функции-члену класса «вид» *OnDraw* передавался указатель на объект «контекст устройства». *OnDraw* выбирала кисть и затем рисовала эллипс. *Контекст устройства* (device context) в Microsoft Windows — ключевой элемент GDI, служащий для представления физического устройства. С каждым объектом «контекст устройства» C++ связан контекст устройства Windows, идентифицируемый 32-разрядным описателем типа HDC.

Библиотека MFC предоставляет несколько классов контекста устройства. Базовый класс *CDC* содержит все необходимые для рисования функции-члены, в том числе несколько виртуальных. Все производные классы, кроме *CMetaFileDC*, от-

личаются только конструкторами и деструкторами. Если вы (или каркас приложения) создали объект производного класса контекста устройства, то указатель на *CDC* можно затем передать функции, например *OnDraw*. Для дисплея обычно применяют производные классы *CClientDC* и *CWindowDC*, для других устройств, таких как принтеры или буферы памяти, — объекты базового класса *CDC*.

«Виртуальность» класса *CDC* — важная особенность каркаса приложения. В главе 17 вы увидите, насколько легко написать код, работающий как с дисплеем, так и с принтером. Например, оператор в *OnDraw*:

```
pDC->TextOut(0, 0, "Hello");
```

посылает текст на дисплей, принтер или в окно предварительного просмотра печати — все определяется классом объекта, на который ссылается параметр *pDC* функции *CView::OnDraw*. Каркас приложения связывает описатели контекста устройства с объектами, представляющими контексты устройств дисплея и принтера. Чтобы связать описатель контекста с объектами, представляющими контексты других устройств, таких как буфер памяти (с ним вы познакомитесь в следующих главах), вы должны после создания объекта вызвать специальную функцию класса.

Классы контекста дисплея *CClientDC* и *CWindowDC*

Как вы помните, в клиентскую область окна не входят рамка, заголовок и меню. Если вы создадите объект *CClientDC*, то получите контекст устройства, представляющий только эту область, — рисовать за ее пределами невозможно. Точка (0, 0) обычно связана с верхним левым углом клиентской области. Как вы увидите, объект *CView* соответствует дочернему окну, содержащемуся в отдельном окне-рамке, зачастую вместе с панелью инструментов, панелью состояния и линейками прокрутки. Все эти окна не входят в клиентскую область окна представления. Если, например, в верхней части окна имеется пристыкованная панель инструментов, то (0, 0) соответствует точке непосредственно под левым краем панели.

Когда вы создаете объект *CWindowDC*, точка (0,0) соответствует левому верхнему краю неклиентской области окна. Этот полнооконный контекст устройства позволяет рисовать по рамке окна, в области заголовка окна и т. п. Не забывайте, что у окна представления нет неклиентской области, поэтому *CWindowDC* более подходит для окон-рамок, а не для окон-представлений.

Создание и уничтожение *CDC*-объектов

Важно своевременно уничтожать созданные объекты *CDC* по окончании работы с ними. Microsoft Windows ограничивает число доступных контекстов устройства, и, если не освободить контекст устройства Windows, небольшой участок памяти будет потерян до завершения программы. Чаще всего объект «контекст устройства» создается в обработчике сообщения, например в *OnLButtonDown*. Проще всего гарантировать уничтожение объекта «контекст устройства» (и освобождение соответствующего контекста устройства Windows), создавая объект в стеке:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;
```

```
   CClientDC dc(this);     // создание контекста устройства (dc) в стеке
   dc.GetClipBox(rect);    // получение ограничивающего прямоугольника
} // контекст устройства автоматически освобождается
```

Заметьте: конструктор объекта *CClientDC* принимает в качестве параметра указатель на окно. Деструктор *CClientDC* вызывается при возврате управления из функции. Вы можете получить указатель на контекст устройства и через функции *CWnd::GetDC*; не забывайте вызывать *ReleaseDC* для освобождения контекста устройства:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CRect rect;

   CDC* pDC = GetDC();           // Указатель на внутренний объект CDC
   pDC->GetClipBox(rect);        // Получение ограничивающего прямоугольника
   ReleaseDC(pDC);               // Не забывайте эту операцию!
}
```

Внимание! Нельзя удалять *CDC*-объект, указатель на который передается функции *OnDraw*, — удалением этого занимается сам каркас приложения.

Состояние контекста устройства

Контекст устройства необходим для рисования. Когда вы используете *CDC*-объект для рисования, скажем, эллипса, полученное на экране (или на принтере) изображение зависит от текущего «состояния» контекста устройства, которое включает:

- связанные с контекстом объекты для рисования: перья, кисти и шрифты;
- режим преобразования координат, определяющий масштаб элементов при их рисовании (мы уже экспериментировали с режимами преобразования координат в главе 5);
- различные детали, например, параметры выравнивания текста и режим заполнения многоугольников.

Мы уже видели, что, если перед рисованием эллипса выбрать, скажем, серую кисть, внутренняя область эллипса закрашивается серым. Вновь созданный контекст устройства имеет некоторые характеристики по умолчанию, в частности черное перо для границ фигур. Остальные характеристики состояния назначаются с помощью функций-членов класса *CDC*. GDI-объекты выбирают для контекста устройства вызовом перегруженных функций *SelectObject*. В любой момент в контексте устройства можно выбрать только одно перо, одну кисть и один шрифт.

Класс *CPaintDC*

Класс *CPaintDC* нужен, только если вы переопределяете функцию *OnPaint* для своего окна представления. Реализация *OnPaint* по умолчанию вызывает *OnDraw* с нужным образом настроенным контекстом устройства, но иногда требуется написать особый код рисования для конкретного дисплея. Особенность *CPaintDC* в том, что

его конструктор и деструктор делают то, что требуется конкретному дисплею. Однако, получив указатель на *CPaintDC*, вы можете использовать его точно так же, как и любой другой контекст устройства. Вот пример функции *OnPaint*, создающей объект *CPaintDC*:

```
void CMyView::OnPaint()
{
    CPaintDC dc(this);
    OnPrepareDC(&dc);        // эта строка объясняется позднее
    dc.TextOut(0, 0, "for the display, not the printer");
    OnDraw(&dc);             // действия, общие для дисплея и принтера
}
```

Для тех, кто программирует в Win32

Конструктор и деструктор *CPaintDC* автоматически вызывают *BeginPaint* и *EndPaint* соответственно. Если контекст устройства создан в стеке, *EndPaint* вызывается автоматически.

Объекты GDI

Каждый тип объектов GDI Windows представлен отдельным классом MFC. *CGdiObject* — абстрактный базовый класс для классов GDI-объектов. Объект GDI представляется объектом класса C++, производного от *CGdiObject*. Вот эти классы.

- ***CBitmap*** (растровое изображение) — массив битов, в котором каждому пикселу дисплея соответствует один или несколько битов. Растровые изображения служат для отображения картинок, а также для создания кистей.
- ***CBrush*** (кисть) — точечный шаблон, используемый для закраски областей.
- ***CFont*** (шрифт) — полный набор символов определенной гарнитуры и размера. Обычно шрифты хранятся на диске как ресурсы, причем некоторые шрифты нужны лишь для определенных устройств.
- ***CPalette*** (палитра) — интерфейс преобразования цветов, позволяющий приложениям в полной мере задействовать цветовые возможности устройства вывода, не мешая другим приложениям.
- ***CPen*** (перо) — инструмент для рисования линий и границ фигур. Можно задать цвет и толщину пера, а также указать тип линии — сплошная, пунктирная или штриховая.
- ***CRgn*** (область) — многоугольник, эллипс или их комбинация. Области позволяют закрашивать, обрезать выводимое изображение и проверять попадание курсора мыши в определенные участки.

Создание и уничтожение GDI-объектов

Мы еще не создавали объектов класса *CGdiObject* — вместо этого мы конструировали объекты производных классов. Конструкторы для некоторых классов, например, *CPen* или *CBrush*, позволяют указать достаточно информации для создания объекта за одну операцию. Создание других объектов, например, *CFont* или *CRgn*,

требует второй операции. Объекты этих классов создаются конструктором по умолчанию, после чего вызывается соответствующая функция, скажем, *CreateFont* или *CreatePolygonRgn*.

У класса *CGdiObject* есть виртуальный деструктор. Деструкторы производных классов удаляют GDI-объекты, связанные с соответствующими объектами C++. Если вы создали объект класса, производного от *CGdiObject*, то обязаны удалить его до завершения программы. Удаляемый GDI-объект надо сначала отделить от контекста устройства. Соответствующий пример мы рассмотрим в следующем разделе.

Для тех, кто программирует в Win32

В Win32 память GDI принадлежит процессу и освобождается при завершении программы. Тем не менее неосвобожденный GDI-объект растрового изображения может напрасно занимать значительный объем памяти.

Управление GDI-объектами

Итак, GDI-объекты нужно удалять, предварительно отсоединив от контекста устройства. Но как? Функции семейства *CDC::SelectObject* выбирают GDI-объект в контекст устройства и возвращают указатель на объект, выбранный в контекст до этого (и теперь отсоединенный). Но возникает проблема: отсоединить старый объект нельзя, не выбрав в контекст нового. Простое решение — сохранить сведения о первоначальном GDI-объекте при выборе своего объекта в контекст и восстановить его по завершении работы. После этого можно удалить свой GDI-объект. Вот пример:

```
void CMyView::OnDraw(CDC* pDC)
{
    CPen newPen(PS_DASHDOT, 2, (COLORREF) 0);     // черное перо шириной 2 пиксела
    CPen* pOldPen = pDC->SelectObject(&newPen);

    pDC->MoveTo(10, 10);
    pDC->LineTo(110, 10);
    pDC->SelectObject(pOldPen);                   // newPen отсоединяется
} // newPen автоматически уничтожается при выходе
```

При уничтожении контекста устройства все его GDI-объекты отсоединяются. Таким образом, если известно, что контекст устройства будет уничтожен до того, как уничтожатся выбранные в него объекты, отсоединять эти объекты не нужно. Так, если вы объявили перо как переменную-член класса «вид» (и инициализировали его при инициализации объекта «вид»), то отсоединять перо внутри *OnDraw* не надо: контекст устройства, жизнью которого управляет обработчик *OnPaint* в базовом классе, будет уничтожен раньше.

Стандартные GDI-объекты

Windows предоставляет ряд *стандартных GDI-объектов* (stock GDI objects). Так как эти объекты — часть Windows, заботиться об их удалении не нужно: Windows игнорирует запросы на удаление стандартных объектов. Функция *CDC::SelectStock-*

Object библиотеки MFC выбирает стандартный объект в контекст устройства и возвращает указатель на выбранный ранее и отсоединяемый от контекста объект. Стандартные объекты удобны, когда нужно отсоединить собственный нестандартный GDI-объект перед его удалением. Стандартный объект можно применять вместо «старого» объекта, который использовался в предыдущем примере[1]:

```
void CMyView::OnDraw(CDC* pDC)
{
   CPen newPen(PS_DASHDOT, 2, (COLORREF) 0);     // черное перо шириной 2 пиксела

   pDC->SelectObject(&newPen);
   pDC->MoveTo(10, 10);
   pDC->LineTo(110, 10);
   pDC->SelectStockObject(BLACK_PEN);            // newPen отсоединяется
} // newPen автоматически уничтожается при выходе
```

Стандартные перья, кисти, шрифты и палитры перечислены в описании функции *CDC::SelectStockObject* в справочнике *Microsoft Foundation Class Reference*.

Время жизни контекста устройства

В случае контекста устройства «дисплей» в начале каждой функции-обработчика сообщения вы получаете «свежий» контекст. Набор выбранных объектов (а также режим преобразования координат и другие параметры контекста) теряется по завершении работы функции. Таким образом, контекст устройства надо всякий раз настраивать заново. Виртуальная функция-член *OnPrepareDC* класса *CView* удобна для установки режима преобразования координат, но своими GDI-объектами вы должны управлять сами.

В контекстах других устройств, таких как принтеры или буферы памяти, назначенные параметры могут сохраняться дольше. С такими «долгожителями» возникают сложности из-за временной природы указателей на объекты С++ контекста устройства, возвращаемых функцией *SelectObject*. (Временный «объект» удаляется каркасом приложения при обработке цикла простоя приложения, некоторое время спустя после возврата управления обработчиком. Подробнее см. раздел MFC Technical Note 3 справочной документации.) Нельзя просто сохранить указатель в переменной-члене класса — вместо этого его нужно преобразовать в описатель Windows (единственный постоянный идентификатор GDI-объекта) с помощью функции-члена *GetSafeHandle*. Вот пример:

```
// m_pPrintFont указывает на объект CFont, созданный в конструкторе CMyView
// m_hOldFont - это переменная-член CMyView типа HFONT, инициализированная нулем

void CMyView::SwtichToCourier(CDC* pDC)
{
   m_pPrintFont->CreateFont(30, 10, 0, 0, 400, FALSE, FALSE,
            0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
            CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
```

[1] Вообще-то отказ от восстановления предыдущего объекта считается признаком плохого стиля программирования. — *Прим. перев.*

```
            DEFAULT_PITCH | FF_MODERN,
            "Courier New");     // шрифт типа True Type
   CFont* pOldFont = pDC->SelectObject(m_pPrintFont);

   // m_hOldFont - открытая переменная-член CGdiObject, в которой хранится описатель
   m_hOldFont = (HFONT) pOldFont->GetSafeHandle();
}

void CMyView::SwitchToOriginalFont(CDC* pDC)
{
   // FromHandle - статическая функция-член,
   // возвращающая указатель на объект
   if (m_hOldFont) {
      pDC->SelectObject(CFont::FromHandle(m_hOldFont));
   }
}

// m_pPrintFont удаляется в деструкторе CMyView
```

Внимание! Будьте осторожны при удалении объекта, указатель на который возвращает *SelectObject*. Если вы создали этот объект сами, то вправе удалить его. Если же указатель временный, что бывает с объектами, первоначально выбранными в контекст устройства, то объект C++ удалять нельзя.

Шрифты

Старым приложениям текстового режима был доступен вывод текста только унылым системным шрифтом. Windows предоставляет массу шрифтов переменного размера, не зависящих от устройств вывода. Шрифты Windows позволяют улучшить внешний вид приложения без особых усилий со стороны программиста. Шрифты TrueType, появившиеся в Windows 3.1, еще эффективнее, и с ними проще работать, чем с зависимыми от устройства шрифтами, которые применялись ранее. Ниже вы познакомитесь с несколькими примерами программ, в которых используются различные шрифты.

Шрифты как GDI-объекты

Поддержка шрифтов встроена в GDI Windows. Это означает, что шрифты — такие же объекты GDI, как другие. Выводимый текст можно масштабировать и обрезать, а шрифты можно выбирать в контекст устройства аналогично перу или кисти. Все правила GDI относительно отсоединения от контекста и удаления применимы и к шрифтам.

Выбор шрифта

Вы можете выбирать из двух типов шрифтов: независимых от устройства (TrueType) и зависимых, таких как System, системный шрифт Windows для дисплея, и шрифт LinePrinter для принтеров LaserJet. Есть и третий путь — задать семейство шриф-

тов и размер, предоставив Windows самой выбрать шрифт. В последнем случае при первой возможности будет выбран шрифт TrueType. В библиотеке MFC есть диалоговое окно выбора шрифта, связанное с выбранным в настоящий момент принтером, поэтому «угадывать» шрифт для принтера практически не надо. Вы позволяете пользователю выбрать для принтера шрифт и его размер и добиваетесь отображения на дисплее, максимально приближенного к этому выбору.

Шрифты и вывод на печать

Для приложений, интенсивно работающих с текстом, размеры шрифта вы скорее всего будете задавать в пунктах (1 пт = 1/72 дюйма). Почему? Большинство, если не все, внутренних шрифтов принтеров определено в терминах пунктов. Так, шрифт LaserJet LinePrinter поставляется с единственным размером 8,5 пт. Шрифты семейства TrueType доступны с произвольным размером в пунктах. При использовании пунктов для преобразования координат вам потребуется соответствующий режим — *MM_TWIPS*. В этом режиме размер 8,5 пт соответствует 170 (8,5×20) твипов — такой размер символа и следует задать.

Отображение шрифтов на дисплее

Если вас не беспокоит точное соответствие изображений на дисплее и на принтере, вы вправе выбрать любой из масштабируемых шрифтов TrueType или же системные шрифты фиксированного размера (стандартные GDI-объекты). При работе со шрифтами TrueType режим преобразования координат не имеет особого значения — просто выберите нужную высоту шрифта. Заботиться о пунктах не нужно.

Подбор соответствия шрифтов принтера, так чтобы изображение на экране точно соответствовало изображению на бумаге, сопряжен с определенными сложностями, однако шрифты TrueType облегчают эту задачу. Но даже если при печати вы используете шрифты TrueType, абсолютного соответствия изображений на дисплее и на бумаге все равно не добиться. Почему? Символы в конечном счете изображаются с помощью пикселов (или точек), и длина строки символов равна сумме ширин отдельных символов в пикселах, возможно, с поправкой на кернинг. Ширина символа в пикселах зависит от шрифта, режима преобразования координат и разрешающей способности устройства вывода. Точное соответствие было бы возможно, только если бы и для принтера, и для дисплея использовался режим *MM_TEXT*, в котором один пиксел в точности равен одной логической единице. Если для вычисления мест переноса строки применить функцию *CDC::GetTextExtent*, места переноса на экране могут иногда отличаться от мест переноса на принтере.

Примечание В режиме предварительного просмотра (Print Preview), который мы рассмотрим в главе 15, переносы на новую строку происходят в тех же местах, что и на бумаге, однако качество отображения в окне предварительного просмотра при этом страдает.

Если вы захотите отобразить текст на экране шрифтом, близким к внутреннему шрифту принтера, технология TrueType облегчит и эту задачу. Windows под-

ставляет наиболее близкий шрифт TrueType. Для шрифта LinePrinter Windows очень точно подставляет свой шрифт Courier New.

Логические и физические дюймы на дисплее

Функция-член *CDC GetDeviceCaps* возвращает параметры дисплея, важные для программирования графики. Шесть следующих параметров (табл. 6-1) предоставляют информацию о размере экрана (указаны значения для типичной видеокарты, сконфигурированной в Windows NT 4.0 для разрешения 640×480 пикселов).

Таблица 6-1. Логические и физические дюймы

Номер параметра	Описание	Значение
HORZSIZE	Физическая ширина в мм	320
VERTSIZE	Физическая высота в мм	240
HORZRES	Ширина в пикселах	640
VERTRES	Высота в растровых линиях	480
LOGPIXELSX	Горизонтальное разрешение в точках на логический дюйм	96
LOGPIXELSY	Вертикальное разрешение в точках на логический дюйм	96

Числа *HORZSIZE* и *VERTSIZE* представляют физические размеры дисплея. (Эти значения могут отличаться от действительных, так как Windows «не знает» размера монитора, подключенного к видеоадаптеру.) Размер дисплея можно вычислить, разделив соответственно *HORZRES* или *VERTRES* на *LOGPIXELSX* или *LOGPIXELSY*. Полученные таким образом размеры называются *логическими* (logical size) дисплея. Используя представленные выше значения и зная, что в дюйме 25,4 мм, можно легко вычислить размеры экрана в разрешении 640×480 пикселов для Windows 2000 и Windows XP. Физический размер дисплея равен 12,60×9,45 дюйма, а логический — 6,67×5 дюймов. Таким образом, физический и логический размеры экрана могут различаться.

В Windows 2000/XP *HORZSIZE* и *VERTSIZE* не зависят от разрешения дисплея, а *LOGPIXELSX* и *LOGPIXELSY* всегда равны 96. Поэтому с изменением разрешения изменяется логический размер, но не физический.

В режиме преобразования координат с фиксированным масштабным множителем, будь то *MM_HIMETRIC* или *MM_TWIPS*, драйвер дисплея использует для пересчета физический размер дисплея. Таким образом, в Windows 2000/XP на мониторе меньшего размера текст также получается меньше, но это не то, чего бы нам хотелось. Напротив, нужно, чтобы размеры шрифта соответствовали логическому, а не физическому размеру дисплея.

Можно создать специальный режим преобразования координат — режим *логических твипов* (logical twips), в котором одна логическая единица равна 1/1440 логического дюйма. Этот режим не зависит от ОС и разрешения монитора и используется такими программами, как Microsoft Word. Следующий код устанавливает режим преобразования в логических твипах:

```
pDC->SetMapMode(MM_ANISOTROPIC);
pDC->SetWindowExt(1440, 1440);
```

```
pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
        - pDC->GetDeviceCaps(LOGPIXELSY));
```

Примечание Как размер шрифта, так и разрешающую способность дисплея можно настроить из Windows Control Panel (Панель управления). Если изменить размер шрифта дисплея со стандартных 100% на 200%, то *HORZSIZE* становится равным 160, *VERTSIZE* — 120, а число точек на дюйм — 192. В этом случае логический размер делится на 2, и размер текста, выводимого в режиме отображения «логические твипы», увеличивается вдвое.

Вычисление высоты символа

CDC-функция *GetTextMetrics* возвращает 5 параметров высоты шрифта, из которых важны только 3 (рис. 6-1): *tmHeight* задает полную высоту шрифта, включая нижние выносные элементы для букв (g, j, p, q и y) и диакритические значки поверх заглавных букв; *tmExternalLeading* задает расстояние между верхом диакритического значка и низом нижнего выносного элемента с предыдущей строки; сумма *tmHeight* и *tmExternalLeading* задает общую высоту символа. Значение *tmExternalLeading* может равняться 0.

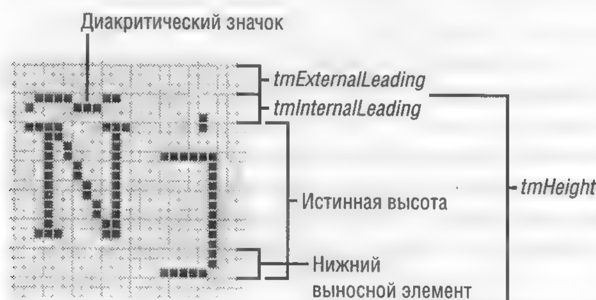


Рис. 6-1. *Размерные параметры шрифта*

Вы можете предположить, что *tmHeight* задает высоту шрифта в пунктах. Это совершенно неверно! Здесь вступает в игру параметр *tmInternalLeading*,, возвращаемый функцией *GetTextMetrics*. Высота в пунктах соответствует разности между *tmHeight* и *tmInternalLeading*. В режиме преобразования координат *MM_TWIPS* выбранный в контекст шрифт в 12 пт может иметь значение *tmHeight*, равное 295, а *tmInternalLeading* — 55 логическим единицам. Тогда истинная высота шрифта, равная 240, соответствует высоте в 12 пт.

Пример Ex06a

В этом примере для окна представления устанавливается режим преобразования координат «логические твипы». Программа выводит текстовую строку TrueType шрифтом Arial высотой 10 пт.

1. **Сгенерируйте проект Ex06a с помощью MFC Application Wizard.** Последовательно выберите в меню File команды New и Project. В качестве типа приложения укажите MFC Application. На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **С помощью окна Properties утилиты Class View переопределите функцию *OnPrepareDC* в классе *CEx06aView*.** Измените код в файле Ex06aView.cpp:

```
void CEx06aView::OnPrepareDC(CDC* pDC, CPrintInfo *pInfo)
{
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1440, 1440);
    pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
                                   - pDC-> GetDeviceCaps(LOGPIXELSY));
}
```

3. **Добавьте закрытую вспомогательную функцию *ShowFont* к классу «вид».** Создайте показанный ниже прототип в Ex06aView.h:

```
private:
    void ShowFont(CDC* pDC, int& nPos, int nPoints);
```

Затем добавьте саму функцию в файл Ex06aView.cpp:

```
void CEx06aView::ShowFont(CDC* pDC, int& nPos, int nPoints)
{
    TEXTMETRIC tm;
    CFont   fontText;
    CString   strText;
    CSize   sizeText;

    fontText.CreateFont(-nPoints * 20, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = (CFont*) pDC->SelectObject(&fontText);
    pDC->GetTextMetrics(&tm);
    TRACE("points = %d, tmHeight = %d, tmInternalLeading = %d,"
        " tmExternalLeading = %d\n", nPoints, tm.tmHeight,
        tm.tmInternalLeading, tm.tmExternalLeading);
    strText.Format("This is %d-point Arial", nPoints);
    sizeText = pDC->GetTextExtent(strText);
    TRACE("string width = %d, string height = %d\n", sizeText.cx,
        sizeText.cy);
    pDC->TextOut(0, nPos, strText);
    pDC->SelectObject(pOldFont);
    nPos -= tm.tmHeight + tm.tmExternalLeading;
}
```

4. **Отредактируйте функцию *OnDraw* в Ex06aView.cpp.** MFC Application Wizard всегда создает для класса «вид» шаблон функции *OnDraw*. Замените ее текст на:

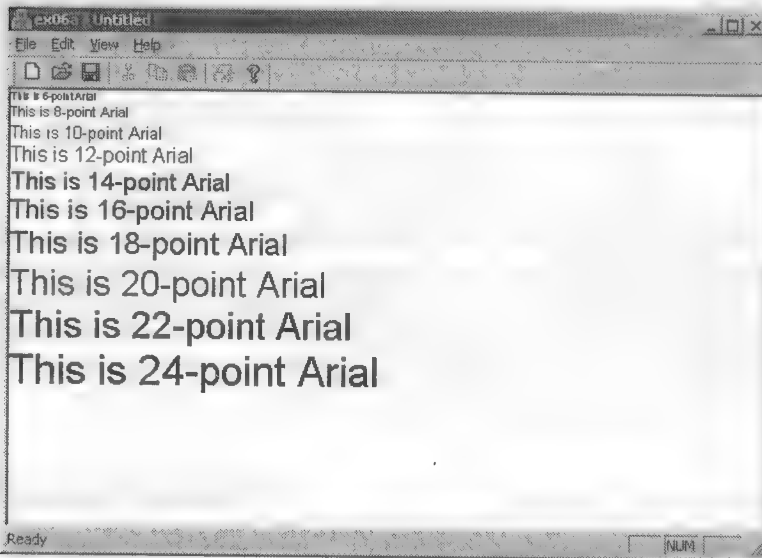
```
void CEx06aView::OnDraw(CDC* pDC)
{
   int nPosition = 0;

   for (int i = 6; i <= 24; i += 2) {
      ShowFont(pDC, nPosition, i);
   }
   TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n",
        pDC->GetDeviceCaps(LOGPIXELSX),
        pDC->GetDeviceCaps(LOGPIXELSY));
   TRACE("HORZSIZE = %d, VERTSIZE = %d\n",
        pDC->GetDeviceCaps(HORZSIZE),
        pDC->GetDeviceCaps(VERTSIZE));
   TRACE("HORZRES = %d, VERTRES = %d\n",
        pDC->GetDeviceCaps(HORZRES),
        pDC->GetDeviceCaps(VERTRES));
}
```

5. **Соберите и запустите программу Ex06a.** Чтобы наблюдать за выводом, генерируемым операторами *TRACE*, программу нужно запускать «в отладчике». Выберите пункт Start из меню Debug в Visual C++ .NET, нажмите клавишу F5 или щелкните на панели инструментов кнопку Continue:



Результат работы программы (если используется стандартная видеоплата VGA) будет примерно таким:



Обратите внимание: размеры выведенных строк не совсем точно соответствуют размерам в пунктах. Несоответствие возникает при преобразовании логических координат в пикселы, выполняемом *подсистемой шрифтов* (font engine). Отла-

дочные данные, фрагмент которых вы видите ниже, показывают параметры шрифтов. (Точные цифры зависят от конкретного драйвера дисплея и видеодрайвера.)

```
points = 6, tmHeight = 150, tmInternalLeading = 30, tmExternalLeading = 4
string width = 990, string height = 150
points = 8, tmHeight = 210, tmInternalLeading = 45, tmExternalLeading = 5
string width = 1380, string height = 210
points = 10, tmHeight = 240, tmInternalLeading = 45, tmExternalLeading = 6
string width = 1770, string height = 240
points = 12, tmHeight = 270, tmInternalLeading = 30, tmExternalLeading = 8
string width = 2130, string height = 270
```

Элементы программы Ex06a

Далее мы обсудим важнейшие элементы программы Ex06a.

Установка режима преобразования координат в функции *OnPrepareDC*

Функцию *OnPrepareDC* вызывает каркас приложения перед вызовом функции *OnDraw*, и потому в ней логичнее всего готовить контекст устройства для *OnDraw*. Если установка режима преобразования координат требуется другим обработчикам сообщений, эти обработчики могут вызывать *OnPrepareDC*.

Закрытая функция-член *ShowFont*

ShowFont содержит код, исполняемый в цикле 10 раз. В программе на С эта функция была бы глобальной, но в С++ лучше сделать ее закрытым членом класса. Иногда такие функции называют *вспомогательными* (helper function).

Функция создает шрифт, выбирает его в контекст устройства, выводит строку в окно и отключает шрифт от контекста устройства. В отладочном варианте программы она также выводит сведения о параметрах шрифта, в том числе истинную длину строки.

Вызов *CFont::CreateFont*

У этой функции масса параметров, но наиболее важны первые два: высота и ширина шрифта. Нулевая ширина означает, что *отношение ширины знаков к высоте* (aspect ratio) шрифта установлено равным значению, определенному разработчиком шрифта в качестве значения по умолчанию. Если указать ненулевое значение, то, как вы увидите в следующем примере, это отношение изменится.

Совет Если нужно получить шрифт точно указанного размера в пунктах, задайте *отрицательное* значение высоты (первый параметр) в вызове функции *CreateFont*. Так, если вывод на принтер осуществляется в режиме преобразования координат *MM_TWIPS*, значение высоты –240 гарантирует высоту шрифта в точности равную 12 пт, а *tmHeight – tmInternalLeading* = 240. Значение +240 даст меньший размер шрифта с *tmHeight*, равным 240.

Последний параметр *CreateFont* задает имя шрифта, в данном случае — Arial (TrueType-шрифт). Если этот параметр равен *NULL*, указание *FF_SWISS* (что означает пропорциональный шрифт без засечек) заставляет Windows выбрать наиболее

подходящий шрифт, которым в зависимости от конкретного размера оказывается System или Arial. Если имя шрифта задано, этот параметр имеет преимущество перед другими. Если, скажем, вместе с Arial указать *FF_ROMAN* (что означает пропорциональный шрифт с засечками), система все равно выберет Arial.

Пример Ex06b

Эта программа похожа на Ex06a, но использует несколько шрифтов. В ней применяется режим преобразования координат *MM_ANISOTROPIC* с масштабом, который изменяется в зависимости от текущих размеров окна. Размеры символов изменяются вместе с размером окна. Программа демонстрирует некоторые шрифты TrueType в сравнении со шрифтами старого типа.

1. **Сгенерируйте проект Ex06b с помощью MFC Application Wizard.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **Используя окно Properties утилиты Class View переопределите функцию *OnPrepareDC* в классе *CEx06bView*.** Измените код в файле Ex06bView.cpp:

```
void CEx06bView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    CRect clientRect;

    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ANISOTROPIC); // ось y направлена вниз
    pDC->SetWindowExt(400, 450);
    pDC->SetViewportExt(clientRect.right, clientRect.bottom);
    pDC->SetViewportOrg(0, 0);
}
```

3. **Добавьте к классу «вид» закрытую вспомогательную функцию *TraceMetrics*.** Добавьте в Ex06bView.h прототип:

```
private:
    void TraceMetrics(CDC* pDC);
```

Затем добавьте саму функцию в Ex06bView.cpp:

```
void CEx06bView::TraceMetrics(CDC* pDC)
{
    TEXTMETRIC tm;
    char    szFaceName[100];

    pDC->GetTextMetrics(&tm);
    pDC->GetTextFace(99, szFaceName);
    TRACE("font = %s, tmHeight = %d, tmInternalLeading = %d,"
        " tmExternalLeading = %d\n", szFaceName, tm.tmHeight,
        tm.tmInternalLeading, .tm.tmExternalLeading);
}
```

4. **Отредактируйте функцию *OnDraw* в Ex06bView.cpp.** MFC Application Wizard всегда создает для класса «вид» шаблон функции *OnDraw*. Замените ее текст на:

```
void CEx06bView::OnDraw(CDC* pDC)
{
   CFont fontTest1, fontTest2, fontTest3, fontTest4;

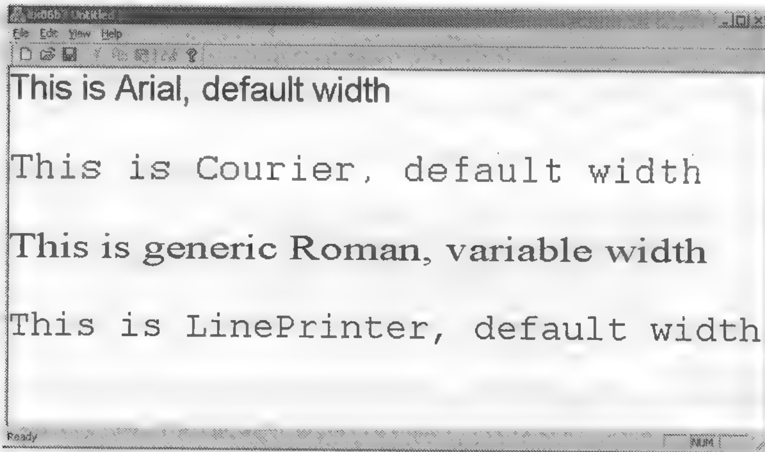
   fontTest1.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                     ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                     DEFAULT_PITCH | FF_SWISS, "Arial");
   CFont* pOldFont = pDC->SelectObject(&fontTest1);
   TraceMetrics(pDC);
   pDC->TextOut(0, 0, " This is Arial, default width ");

   fontTest2.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                     ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                     DEFAULT_PITCH | FF_MODERN, "Courier");
                     // шрифт не из семейства TrueType
   pDC->SelectObject(&fontTest2);
   TraceMetrics(pDC);
   pDC->TextOut(0, 100, "This is Courier, default width");

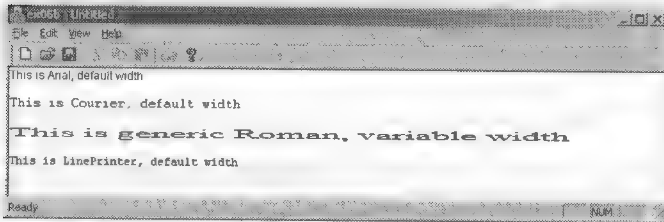
   fontTest3.CreateFont(50, 10, 0, 0, 400, FALSE, FALSE, 0,
                     ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                     DEFAULT_PITCH | FF_ROMAN, NULL);
   pDC->SelectObject(&fontTest3);
   TraceMetrics(pDC);
   pDC->TextOut(0, 200, "This is generic Roman, variable width");

   fontTest4.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                     ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                     DEFAULT_PITCH | FF_MODERN, "LinePrinter");
   pDC->SelectObject(&fontTest4);
   TraceMetrics(pDC);
   pDC->TextOut(0, 300, "This is LinePrinter, default width");
   pDC->SelectObject(pOldFont);
}
```

5. **Соберите и запустите программу Ex06b.** Чтобы видеть вывод, генерируемый операторами TRACE, запустите программу «из-под отладчика». Окно программы показано на рисунке.



Уменьшите размеры окна и проследите, как меняется размер шрифтов. Сравните это окно с показанным выше.



Продолжайте уменьшать окно: размер шрифта Courier по достижении некоторого минимума перестает меняться. Последите также за изменением ширины шрифта Roman.

Элементы программы Ex06b

Далее обсуждаются важнейшие элементы примера Ex06b.

Функция-член *OnDraw*

Эта функция выводит в окне строки с использованием четырех разных шрифтов:

- *fontTest1* — шрифт Arial типа TrueType со стандартной шириной;
- *fontTest2* — шрифт Courier старого типа со стандартной шириной; обратите внимание на неровные края букв при большом размере шрифта;
- *fontTest3* — типовой шрифт Roman, вместо которого Windows использует TrueType-шрифт Times New Roman с программным выбором ширины; ширина связана с горизонтальным размером окна — шрифт будет растягиваться так, чтобы строка занимала всю ширину окна;
- *fontTest4* — в программе задан шрифт LinePrinter, но, поскольку он не является шрифтом Windows для дисплея, подсистема шрифтов, руководствуясь параметром *FF_MODERN*, выберет TrueType-шрифт Courier New.

Вспомогательная функция *TraceMetrics*

Вызвав *CDC::GetTextMetrics* и *CDC::GetTextFace*, эта функция получает параметры текущего шрифта, которые затем отображает в окне отладки.

Пример Ex06c: снова *CScrollView*

Мы уже встречались с классом *CScrollView* в главе 5 (пример Ex05c). В Ex06c используется окно с прокруткой и режимом преобразования координат *MM_LOENGLISH*, что позволяет пользователю перемещать эллипс мышью, применяя «захват» мыши. Прокрутка с клавиатуры не реализована, но вы можете добавить ее, «одолжив» функцию *OnKeyDown* из Ex05c.

Вместо стандартной кисти мы задействуем для эллипса *трафаретную кисть* (pattern brush) — настоящий GDI-объект. С этим связана одна сложность: после прокрутки окна нужно переустанавливать начальную точку, иначе полосы трафарета не совпадут, и результат будет выглядеть довольно уродливо.

Как и в Ex05c, здесь используется класс «вид», производный от *CScrollView*.

1. **С помощью MFC Application Wizard создайте проект Ex06c.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Не забудьте выбрать в качестве базового класса *CScrollView*.
2. **Отредактируйте заголовок класса *CEx06cView* в файле Ex06cView.h.** В объявление класса *CEx06cView* добавьте строки:

```
private:
    const CSize m_sizeEllipse;    // логические
    CPoint m_pointTopLeft;        // логические, верхний левый угол
                                  // прямоугольника эллипса
    CSize m_sizeOffset;           // физические, смещение от верхнего левого
                                  // угла прямоугольника до точки захвата мыши
    BOOL m_bCaptured;
```

3. **В окне Class View выберите класс *CEx06cView* и в окне Properties добавьте в него три обработчика событий.** Добавьте обработчики:

Сообщение	Функция-член
WM_LBUTTONDOWN	*OnLButtonDown*
WM_LBUTTONUP	*OnLButtonUp*
WM_MOUSEMOVE	*OnMouseMove*

4. **Отредактируйте функции-обработчики сообщений в классе *CEx06cView*.** Шаблоны этих функций сгенерировал мастер Class View на предыдущем этапе. Найдите эти функции в Ex06cView.cpp и поместите в них код:

```
void CEx06cView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rectEllipse(m_pointTopLeft, m_sizeEllipse);    // все еще логические
                                                         // координаты
    CRgn circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // теперь в аппаратных координатах
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
```

```
        // захватывая мышь, мы уверены в последующем появлении сообщения LButtonUp
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // аппаратные координаты
        // на время захвата мыши изменяем ее курсор
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
    }
}

void CEx06cView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_bCaptured) {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}

void CEx06cView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (m_bCaptured) {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}
```

5. **Отредактируйте конструктор класса *CEx06cView,* функции *OnDraw* и *OnInitialUpdate.*** Шаблоны этих функций сгенерированы MFC Application Wizard. Найдите их в Ex06cView.cpp и поместите в них такой код.

```
CEx06cView::CEx06cView() : m_sizeEllipse(100, -100),
                           m_pointTopLeft(0, 0),
                           m_sizeOffset(0, 0)
{
    m_bCaptured = FALSE;
}

void CEx06cView::OnDraw(CDC* pDC)
{
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    CPoint point(0, 0);              // логическая точка (0, 0)

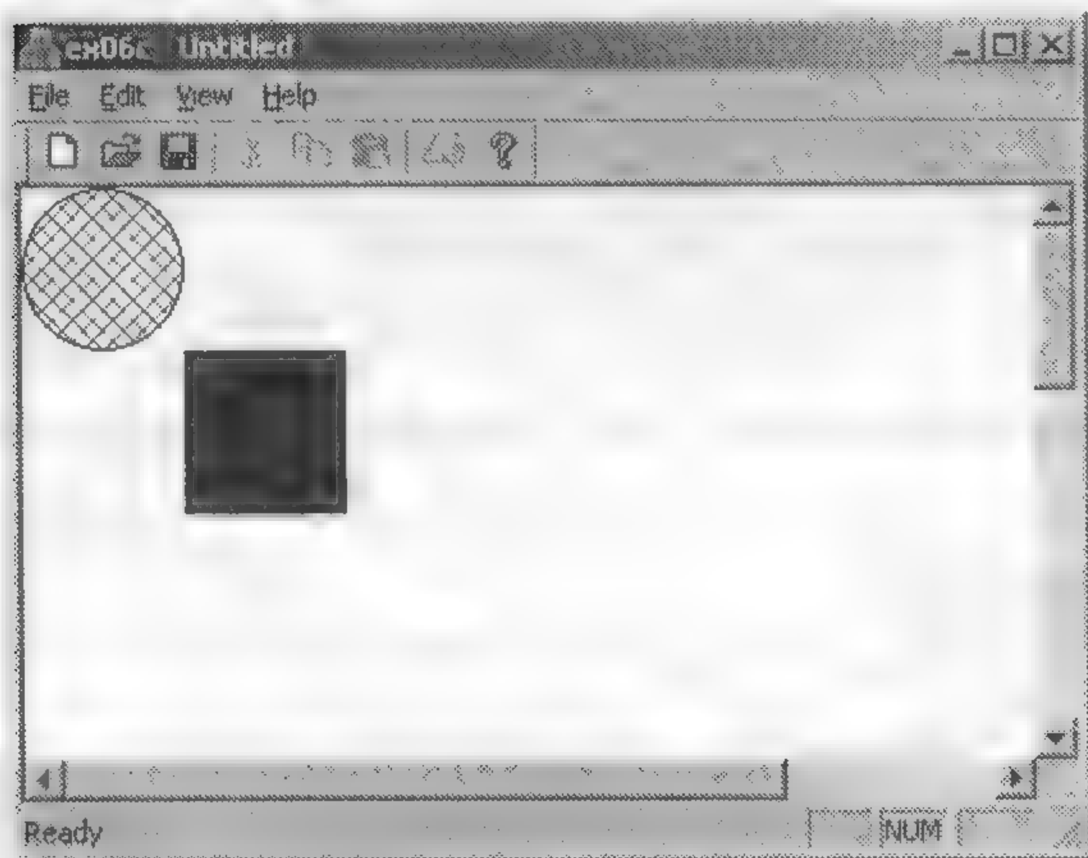
    pDC->LPtoDP(&point);             // переход к аппаратным координатам
```

```
    pDC->SetBrushOrg(point);           // для выравнивая трафарета кисти
                                       // по начальной точке окна
    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH);          // отсоединяем brushHatch
    pDC->Rectangle(CRect(100, -100, 200, -200));  // объявляем прямоугольник
                                                  // недействительным
}

void CEx06cView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    CSize sizeTotal(800, 1050);        // 8x10,5 дюймов
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
}
```

6. **Соберите и запустите программу Ex06c.** Программа поддерживает перетаскивание эллипса мышью и прокрутку изображения в окне. Окно программы должно выглядеть примерно, как на рисунке. При перемещении эллипса следите за черным прямоугольником. Вы должны заметить характерное поведение, когда изображение прямоугольника становится «недействительным».



Элементы программы Ex06c

Ниже обсуждаются важнейшие элементы примера Ex06c.

Переменные-члены *m_sizeEllipse* и *m_pointTopLeft*

Вместо того чтобы хранить ограничивающий прямоугольник эллипса в одном объекте *CRect*, программа раздельно хранит размер (*m_sizeEllipse*) и положение левого верхнего угла этого прямоугольника (*m_pointTopLeft*). Чтобы переместить эллипс, программе достаточно заново вычислить *m_pointTopLeft*, при этом погрешность округления не влияет на размер эллипса.

Переменная-член *m_sizeOffset*

Когда функция *OnMouseMove* перемещает эллипс, относительное положение указателя мыши внутри эллипса должно оставаться неизменным с того момента, когда пользователь нажал левую кнопку. В переменной *m_sizeOffset* хранится первоначальное смещение мыши относительно левого верхнего угла прямоугольника эллипса.

Переменная-член *m_bCaptured*

Эта логическая переменная устанавливается в TRUE на время выполнения операции перемещения эллипса мышью.

Функции *SetCapture* и *ReleaseCapture*

Функция *SetCapture* класса *CWnd* захватывает мышь, после чего сообщения о перемещении мыши посылаются только в данное окно, даже если указатель мыши выйдет за его пределы. Нежелательный побочный эффект состоит в том, что эллипс может выйти за пределы окна и «потеряться». Желательный и необходимый эффект — то, что *все* последующие сообщения мыши, в том числе *WM_LBUTTONUP* (которое иначе было бы потеряно), теперь посылаются нашему окну. Функция Win32 *ReleaseCapture* снимает захват мыши.

Win32-функции *SetCursor* и *LoadCursor*

Некоторые функции Win32 не имеют эквивалентной «обертки» в библиотеке MFC. По соглашению мы используем оператор разрешения области действия C++ (::) при вызове функций Win32 напрямую. В этом случае вероятность конфликта имен с функцией-членом класса *CView* исключена, но всегда можно вместо функции-члена класса вызвать одноименную функцию Win32. При этом оператор :: гарантирует, что вызывается именно глобальная Win32-функция.

Если первый параметр равен NULL, *LoadCursor* создает *ресурс-курсор* (cursor resource) из заданного стандартного курсора мыши Windows. Функция *SetCursor* активизирует заданный ресурс курсора. Этот курсор остается активным, пока мышь захвачена.

Функция-член *CScrollView::OnPrepareDC*

Виртуальная функция *OnPrepareDC* в классе *CView* ничего не делает. В классе *CScrollView* она переопределена и устанавливает режим преобразования и точку начала координат окна «вид» на основании параметров, переданных *SetScrollSizes* в функции *OnInitialUpdate*. Каркас приложения автоматически вызывает *OnPrepareDC* перед вызовом *OnDraw*, так что об этом заботиться не надо. Из любого другого обработчика сообщения, использующего контекст устройства окна представления, например, из *OnLButtonDown* и *OnMouseMove*, функцию *OnPrepareDC* придется вызывать самостоятельно.

Преобразование координат в *OnMouseMove*

Эта функция содержит несколько операторов преобразования. Алгоритм таков.

1. Создать предыдущий описывающий прямоугольник эллипса и преобразовать его из логических координат в аппаратные.
2. Сделать предыдущий прямоугольник недействительным.

3. Обновить координаты левого верхнего угла прямоугольника.
4. Создать новый прямоугольник и преобразовать его в аппаратные координаты.
5. Сделать недействительным новый прямоугольник.

Функция вызывает *InvalidateRect* дважды. Windows «накапливает» недействительные прямоугольники и вычисляет новый недействительный прямоугольник, представляющий собой пересечение объединения двух первых с клиентским прямоугольником.

Функция *OnDraw*

Вызов *SetBrushOrg* обеспечивает правильную перерисовку трафарета, применяемого для внутренней области эллипса, когда пользователь прокручивает изображение в окне. Кисть выравнивается относительно точки, находящейся в левом верхнем углу логического окна и преобразованной в аппаратные координаты. Это важное исключение из правила, по которому параметрами функций-членов *CDC* служат логические координаты.

Режим *SetScaleToFitSize* класса *CScrollView*

Класс *CScrollView* поддерживает масштабирование по размеру окна. В этом режиме вся область прокрутки отображается в окне целиком. Применяется режим преобразования координат *MM_ANISOTROPIC* с одним ограничением: положительные значения координаты *y* всегда возрастают при движении вниз, как в режиме *MM_TEXT*.

Чтобы задействовать масштабирование по размеру окна, замените вызов *SetScrollSizes* на:

```
SetScaleToFitSize(sizeTotal);
```

Вы можете вызывать эту функцию по команде меню Shrink to fit («сжать по размеру окна») и реализовать таким образом переключение между режимом прокрутки и режимом масштабирования по размеру окна.

Режим преобразования координат «логический твип» в окне с прокруткой

MFC-класс *CScrollView* позволяет прибегать только к стандартным режимам преобразования координат. В примере Ex17a в главе 17 представлен новый класс *CLogScrollView*, поддерживающий режим «логические твипы».

Растровые изображения

Без графических изображений Windows-программы выглядели бы довольно уныло. Иногда приложение вообще бесполезно без графики, но с рисунками любая программа становится гораздо интереснее. *Растровые изображения*, или *битовые карты* (bitmaps), Windows — это массивы битов, представляющих растровые точки (пикселы) дисплея. Кажется, все очень просто, но, прежде чем применять растровые изображения для создания Windows-приложений профессионального уровня, вам придется многому научиться.

В этом разделе вы узнаете, как создавать *аппаратно-независимые растровые изображения* (Device-Independent Bitmaps, DIBs). DIB упрощает работу с цветами и принтером, а иногда позволяет добиться и более высокой производительнос-

ти. Новая Win32-функция *CreateDIBSection* предоставляет все преимущества DIB в сочетании с возможностями растрового изображения GDI.

Вы также узнаете, как размещать растровые изображения на командных кнопках, применяя MFC-класс *CBitmapButton*. Эта часть не связана с DIB, но прием этот очень полезен, и им очень трудно овладеть без примера.

Растровые изображения GDI и DIB

Этот раздел посвящен в основном аппаратно-независимым растровым изображениям (DIB). Намного больше о работе с ними вы узнаете из справочной системы MSDN комплекта ресурсов Platform SDK. В Windows два типа растровых изображений: GDI-растр и DIB. Растровые изображения GDI используются уже давно, и за более подробной информацией о них обратитесь к соответствующей литературе.

Класс *CBitmap* библиотеки MFC представляет объект «растровое изображение GDI». С этим объектом связана аппаратно-зависимая структура данных Windows, внутренняя для модуля GDI. Ваша программа может получить копию данных объекта, но структура битов в ней зависит от конкретной аппаратуры дисплея. Растровые изображения GDI можно свободно передавать между разными программами на одном компьютере, но из-за аппаратной зависимости копировать их на другой компьютер бессмысленно.

Растровое изображение GDI — это лишь один из GDI-объектов, таких как перо или шрифт. Поэтому вам достаточно создать растровое изображение, а затем выбрать его в контекст устройства. Закончив работу с растровым изображением, вы должны отключить его от контекста и удалить. Впрочем, это вы уже знаете.

Однако есть одна особенность, связанная с тем, что «растровое изображение» на экране и принтере в действительности представляет собой поверхность экрана или бумагу. Растровое изображение нельзя напрямую выбрать в аппаратный контекст дисплея или принтера — нужно позаботиться о создании специального ***контекста устройства в памяти*** (memory device context) для растрового изображения. Для этого служит функция *CDC::CreateCompatibleDC*, а затем вызывается функция-член *StretchBlt* или *BitBlt* класса *CDC*, которая копирует контекст устройства в памяти в контекст «реального» устройства. Такие «преобразующие» функции обычно вызываются в функции *OnDraw* класса «вид». Естественно, по завершении работы вы должны не забыть очистить контекст устройства в памяти.

Для тех, кто программирует в Win32

В Win32 можно поместить описатель растрового изображения GDI в глобальный буфер обмена для передачи другому процессу, но Windows при этом автоматически преобразует аппаратно-зависимые данные в DIB, который и копирует в совместно используемую память. Это еще один аргумент в пользу DIB.

DIB-изображения имеют ряд преимуществ в сравнении с GDI-растрами. Поскольку DIB содержит сведения о цвете, упрощается работа с цветовой палитрой, а также управление градациями серого цвета при печати. Любой компьютер под

управлением Windows способен обрабатывать DIB, которые обычно хранятся в виде дисковых BMP-файлов или как ресурсы в EXE- и DLL-файлах. В частности, фоновая картинка на вашем рабочем столе считывается при запуске Windows из BMP-файла. Основным форматом хранения данных для программы Windows Paint служит BMP, а Visual C++ .NET использует BMP-файлы для хранения изображений, например кнопок панелей инструментов. Есть и другие форматы обмена графическими данными, например, TIFF, GIF или JPEG, но Win32 API напрямую поддерживает только формат DIB.

Цветные и монохромные растровые изображения

Существуют небольшие различия в обработке ОС Windows цветных растров и цветов кисти. Многие растровые изображения — 16-цветные. У обычной VGA-карты 4 цветовых «плоскости», и для представления пиксела используется по одному биту в каждой. При формировании растрового изображения задаются 4-разрядные значения цвета. Таким образом, для обычной VGA-карты цвета растрового изображения ограничены 16 стандартными цветами. *Смешанные* (dithered) цвета в растровых изображениях не применяются.

В монохромном растровом изображении лишь одна плоскость. Каждый пиксел представляется одним битом, значения которого принимают одно из двух значений: 1 (включен) или 0 (выключен). Цвет текста задает функция *CDC::SetTextColor*, а фона — функция *SetBkColor*. Оба цвета позволяет определить Windows-макрос *RGB*.

DIB-изображения и класс *CDib*

В MFC есть класс *CBitmap* для GDI-растров, но класса для DIB нет. Не волнуйтесь, вы его получите. Это полностью переработанный класс *CDib* из предыдущих изданий этой книги (до 4-го издания). В нем задействованы все преимущества таких средств Win32, как проецирование файлов в память, улучшенное управление памятью и DIB-секции. Включена и поддержка палитр. Однако прежде, чем изучать класс *CDib*, познакомимся с DIB поближе.

Несколько слов о программировании палитры

Программировать палитру в Windows очень сложно, но дело с ней придется иметь, пока есть пользователи, которые работают с дисплеями в режиме 8 бит/пиксел, что приходится делать, когда объем памяти на видеокарте не превышает 1 Мб.

Допустим, мы воспроизводим в окне одно DIB-изображение. Первое, что надо сделать, — создать *логическую палитру* (logical palette) — GDI-объект, содержащий цвета из DIB. Затем нужно реализовать ее в аппаратную *системную палитру* (system palette) — таблицу из 256 цветов, одновременно отображаемых видеокартой. Если программа активна (foreground), Windows пытается скопировать все заданные цвета в системную палитру, но не трогает 20 стандартных цветов, всегда присутствующих в ней. Обычно DIB выглядит именно так, как задумано.

А если активна (на переднем плане) другая программа, которая отображает DIB с лесным пейзажем в 236 оттенков зеленого? Наша программа по-прежнему реализует свою палитру, но на этот раз несколько иначе. Теперь системная палитра

не меняется, а просто палитры — системная и наша логическая — сопоставляются по-новому. Например, если наша программа отображает цвет розового неона, Windows сопоставит его стандартному красному цвету. И если вы забудете в программе реализовать собственную палитру, при активизации другой программы весь розовый неон позеленеет.

Пример с лесным пейзажем, конечно, крайность, так как в нем предполагается, что другая программа отбирает у нас 236 цветов. Если же она реализует логическую палитру всего лишь из 200 цветов, Windows позволит нашей программе загрузить 36 своих цветов, в том числе, хочется верить, и розовый неон.

Итак, когда же программе реализовать собственную палитру? Сообщение Windows *WM_PALETTECHANGED* направляется в основное окно вашей программы всякий раз, когда какая-то программа (и ваша тоже) реализует свою палитру. Другое сообщение, *WM_QUERYNEWPALETTE*, посылается, когда одно из окон программы получает фокус ввода. Программа должна реализовать свою палитру в ответ на оба этих сообщения (если только не она их отправила). Однако эти сообщения не передаются окну объекта «вид». Вы должны создать соответствующие обработчики в основном окне приложения и предусмотреть уведомление объекта «вид». Взаимосвязи окна-рамки и объекта «вид» обсуждаются в главе 14.

Для реализации палитры вызывают Win32-функцию *RealizePalette*, но сначала надо вызвать *SelectPalette*, чтобы подключить логическую палитру DIB-изображения к контексту устройства. У *SelectPalette* есть параметр — флаг, который в обработчиках сообщений *WM_PALETTECHANGED* и *WM_QUERYNEWPALETTE* обычно устанавливается в *FALSE*. Это гарантирует реализацию палитры как палитры переднего плана, если ваше приложение действительно активно. Установив флаг в *TRUE*, вы заставите Windows реализовать палитру так, будто приложение исполняется в фоновом режиме.

Вы также должны вызывать *RealizePalette* для каждого DIB-изображения, воспроизводимого при работе функции *OnDraw*. Сначала, конечно, нужно вызвать *SelectPalette*, на этот раз с флажком *TRUE*. Если программа показывает несколько DIB-изображений, каждое со своей палитрой, задача существенно усложняется. Но в принципе нужно выбрать палитру для одного из DIB-изображений и реализовать ее (подключив с параметром *FALSE*) в обработчиках сообщений для палитры. Это DIB-изображение скорее всего будет выглядеть лучше других. Есть несколько способов совмещения палитр, но куда проще пойти и купить дополнительную видеопамять.

DIB-растры, пикселы и цветовые таблицы

DIB-растр состоит из двумерного массива элементов — *пикселов* (pixel). Часто каждый пиксел DIB соответствует пикселу на экране, но его можно также спроецировать и на какую-то логическую область экрана — в зависимости от режима преобразования координат и параметров масштабирования функции вывода изображения.

Пиксел состоит из 1, 4, 8, 16, 24 или 32 последовательных битов — все определяется цветовым разрешением конкретного изображения. В DIB с разрешениями 16, 24 и 32 бит/пиксел каждый пиксел представляет какой-то RGB-цвет. В DIB с разрешением 16 бит/пиксел на значения красного, зеленого и голубого цветов,

как правило, отводится по 5 бит, а в более распространенных DIB с разрешением 24 бит/пиксел — по 8. DIB с разрешением 16 и 24 бит/пиксел оптимизированы для видеоплат, способных воспроизводить одновременно 65 536 или 16,7 миллионов цветов соответственно.

DIB-изображения с разрешением 1 бит/пиксел — монохромные, но не обязательно черно-белые. Допускаются любые 2 цвета из встроенной в DIB цветовой таблицы. В цветовой таблице монохромного растрового изображения есть два 32-разрядных элемента, каждый из которых содержит по 8 битов на красный, зеленый и голубой, и еще 8 битов для флагов. Для пикселов со значением 0 используется первый элемент, а со значением 1 — второй. Если ваша видеоплата воспроизводит 65 536 или 16,7 миллиона цветов, Windows сможет напрямую отобразить эти два цвета. (В видеорежиме с 65 536 цветами Windows урезает 8-битные значения до 5-битных.) Если видеоплата работает в режиме 256 цветов с использованием палитры, программа сможет скорректировать системную палитру, загрузив в нее нужные два цвета.

Распространены также DIB с цветовым разрешением 8 бит/пиксел. Как и в монохромных DIB, у них тоже есть цветовая таблица, но с 256 (или менее) 32-разрядными элементами. Каждый пиксел в этой таблице — индекс. При наличии видеоплаты с поддержкой палитр программа может создать из этих 256 элементов логическую палитру. Если системную палитру контролирует другая программа, выполняемая на переднем плане, Windows старается наилучшим образом сопоставить цвета вашей логической палитры с цветами системной.

А если попытаться воспроизвести DIB с цветовым разрешением 24 бит/пиксел на 256-цветной видеоплате с поддержкой палитр? В идеале автор DIB должен был бы включить в нее цветовую таблицу с важнейшими цветами. В этом случае ваша программа может построить по этой таблице логическую палитру, и изображение выглядело бы нормально. Если же цветовой таблицы в DIB нет, используйте палитру, возвращаемую функцией Win32 *CreateHalftonePalette*. В любом случае это лучше, чем 20 стандартных цветов, доступных при отсутствии какой бы то ни было палитры. Еще один вариант — проанализировать DIB, чтобы определить важнейшие цвета, но для этого можно приобрести готовую утилиту.

Структура DIB в BMP-файле

Как вы помните, DIB — это стандартный формат Windows для растровых изображений, и хранятся они в BMP-файлах. Заглянем же внутрь BMP-файла и посмотрим, что там (рис. 6-2).

Структура *BITMAPFILEHEADER* содержит смещение битов изображения, на основе которого можно вычислить общий размер структуры *BITMAPINFOHEADER* и следующей за ней цветовой таблицы. В структуре *BITMAPFILEHEADER* есть поле, хранящее размер файла, но полагаться на него не стоит, так как вам неизвестны единицы измерения: байты, слова или двойные слова[1].

Структура *BITMAPINFOHEADER* определяет размеры растрового изображения, число битов на пиксел, информацию об упаковке изображений с 4 и 8 бит/пик-

---

[1] Автор ошибается: согласно документации Windows размер файла указывается в байтах. — *Прим. перев.*

сел, а также количество элементов в цветовой таблице. Если DIB упакован, этот заголовок содержит размер массива пикселов — иначе этот размер можно вычислить по размеру растрового изображения и цветовому разрешению. За заголовком — цветовая таблица (если она вообще есть в данном DIB). Далее размещается собственно DIB-изображение, состоящее из пикселов, упорядоченных по столбцам в пределах строк, начиная с нижней. Каждая строка выравнивается по границе, кратной 4 байтам.

Единственное место, где вы встретите структуру *BITMAPFILEHEADER*, — BMP-файл. Если DIB получен, скажем, из буфера обмена, то заголовка файла в этой структуре нет. Цветовая таблица располагается за структурой *BITMAPINFOHEADER*, но изображение не всегда располагается сразу за цветовой таблицей. Поэтому, если вы, допустим, используете функцию *CreateDIBSection*, выделяйте место для заголовка растрового изображения и цветовой таблицы, а уж где размещать изображение, пусть решает сама Windows.

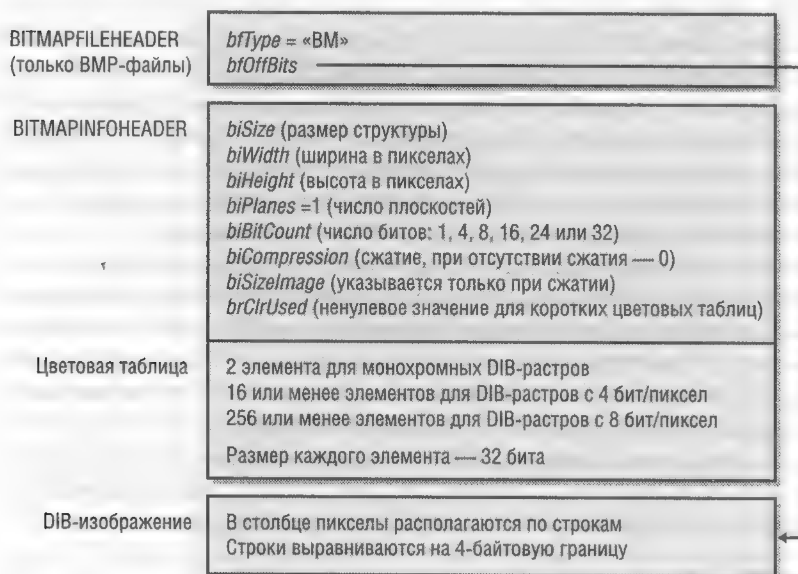


Рис. 6-2. *Формат BMP-файла*

Функции для работы с DIB

В Windows предусмотрено несколько важных функций для работы с DIB. Ни для одной из них нет аналога в MFC, поэтому более подробную информацию ищите в документации по Win32. Мы приведем лишь краткое описание данных функций.

- ***SetDIBitsToDevice*** непосредственно отображает DIB на экране или принтере. Масштабирование не осуществляется; один пиксел растрового изображения соответствует одному пикселу на экране или на принтере. Отсутствие масштабирования снижает полезность этой функции. Механизм ее работы отличен от *BitBlt*, в которой используются логические координаты.
- ***StretchDIBits*** непосредственно отображает DIB на экране или принтере; работает аналогично *StretchBlt*.

- ***GetDIBits*** создает DIB из GDI-растра, используя выделенную программистом память. Вы можете в какой-то степени управлять форматом DIB, так как эта функция позволяет задавать число «цветовых» битов на пиксел и сжатие. Если применяется сжатие, *GetDIBits* надо вызывать дважды: сначала для вычисления нужного объема памяти, а затем — для генерации данных DIB.
- ***CreateDIBitmap*** создает GDI-растр из DIB. Как и все подобные DIB-функции, требует передачи указателя на контекст устройства в качестве параметра. Можно указывать контекст дисплея; использовать контекст устройства в памяти не обязательно.
- ***CreateDIBSection*** создает особую разновидность DIB — *DIB-секцию* (DIB section) и возвращает описатель GDI-растра. Эта функция сочетает в себе лучшие средства работы с DIB и растровыми изображениями GDI. Вы получаете прямой доступ к памяти DIB и, имея описатель растрового изображения и контекст устройства в памяти, можете вызывать GDI-функции для рисования в DIB.

Класс *CDib*

Не пугайтесь сложностей работы с DIB — класс *CDib* упростит ее. Чтобы познакомиться с ним, лучше всего изучить его открытые члены — функции и переменные. Ниже показан заголовочный файл класса *CDib*. Код реализации вы найдете в папке Ex06d на компакт-диске.

CDib.H

```
#ifndef _INSIDE_VISUAL_CPP_CDIB
#define _INSIDE_VISUAL_CPP_CDIB

class CDib : public CObject
{
   enum Alloc {noAlloc, crtAlloc,
               heapAlloc};          // относится к BITMAPINFOHEADER
   DECLARE_SERIAL(CDib)
public:
   LPVOID m_lpvColorTable;
   HBITMAP m_hBitmap;
   LPBYTE m_lpImage;                // начальный адрес битов DIB
   LPBITMAPINFOHEADER m_lpBMIH;     // буфер, содержащий BITMAPINFOHEADER

private:
   HGLOBAL   m_hGlobal;    // описатель может выделяться этим классом или вне
                           // его, в последнем случае его нужно освободить
   Alloc     m_nBmihAlloc;
   Alloc     m_nImageAlloc;
   DWORD     m_dwSizeImage;         // размер массива битов,
                                    // а не структуры BITMAPINFOHEADER
                                    // или BITMAPFILEHEADER
   int       m_nColorTableEntries;
```

см. след. стр.

```
    HANDLE   m_hFile;
    HANDLE   m_hMap;
    LPVOID   m_lpvFile;
    HPALETTE m_hPalette;
public:
    CDib();
    CDib(CSize size, int nBitCount);  // создает BITMAPINFOHEADER
    ~CDib();
    int GetSizeImage() {return m_dwSizeImage;}
    int GetSizeHeader()
        {return sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries;}
    CSize GetDimensions();
    BOOL AttachMapFile(const char* strPathname, BOOL bShare = FALSE);
    BOOL CopyToMapFile(const char* strPathname);
    BOOL AttachMemory(LPVOID lpvMem, BOOL bMustDelete = FALSE,
            HGLOBAL hGlobal = NULL);
    BOOL Draw(CDC* pDC, CPoint origin,
            CSize size);             // пока не реализована CreateDIBSection
    HBITMAP CreateSection(CDC* pDC = NULL);
    UINT UsePalette(CDC* pDC, BOOL bBackground = FALSE);
    BOOL MakePalette();
    BOOL SetSystemPalette(CDC* pDC);
    BOOL Compress(CDC* pDC,
            BOOL bCompress = TRUE);   // если FALSE, то распаковать
    HBITMAP CreateBitmap(CDC* pDC);
    BOOL Read(CFile* pFile);
    BOOL ReadSection(CFile* pFile, CDC* pDC = NULL);
    BOOL Write(CFile* pFile);
    void Serialize(CArchive& ar);
    void Empty();
private:
    void DetachMapFile();
    void ComputePaletteSize(int nBitCount);
    void ComputeMetrics();
};
#endif   _INSIDE_VISUAL_CPP_CDIB
```

Далее кратко описаны функции-члены *CDib*, начиная с конструкторов и деструктора.

- **Конструктор по умолчанию** применяется, если предполагается загружать DIB из файла или подсоединять экземпляр класса к уже существующему в памяти DIB. Конструктор по умолчанию создает пустой DIB-объект.
- **Конструктор DIB-секции** применяется, если нужна DIB-секция, создаваемая при использовании *CreateDIBSection*. Его параметры определяют размер и количество цветов в DIB. Конструктор выделяет память для хранения заголовка, но не для хранения изображения. Этот конструктор также можно применять для самостоятельного выделения памяти для изображения.

Параметр	Описание
size	Объект *CSize*, задающий ширину и высоту DIB
nBitCount	Число бит/пиксел. Допустимые значения: 1, 4, 8, 16, 24 или 32

- **Деструктор** освобождает всю выделенную для DIB память.
- ***AttachMapFile*** открывает проецируемый в память файл в режиме чтения и подсоединяет его к объекту *CDib*. Функция сразу возвращает управление, так как не считывает файл в память, пока тот не потребуется. Но при доступе к DIB могут возникать задержки, связанные с подкачкой файла в память. Функция освобождает выделенную ранее память и, если надо, закрывает присоединенный ранее проецируемый файл.

Параметр	Описание
strPathname	Полное имя (с путем) проецируемого файла
bShare	Флаг, значение которого равно *TRUE*, если файл надо открыть в режиме совместного использования; по умолчанию — *FALSE*
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***AttachMemory*** связывает существующий объект *CDib* с DIB в памяти. Эта память может быть ресурсом программы или находиться в буфере обмена или в объекте данных OLE. Ее можно выделить оператором *new* или функцией *GlobalAlloc*.

Параметр	Описание
lpvMem	Адрес присоединяемой памяти
bMustDelete	Флаг, значение которого равно *TRUE*, если за освобождение памяти отвечает объект *CDib*; по умолчанию — *FALSE*
hGlobal	Если память получена вызовом функции Win32 *GlobalAlloc*, объект *CDib* должен запомнить описатель области памяти, чтобы впоследствии освободить ее (при *bMustDelete*, равном *TRUE*)
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***Compress*** восстанавливает DIB в сжатом или развернутом виде, т. е. преобразует существующее DIB-изображение в GDI-растр, после чего повторно создает сжатое или развернутое DIB-изображение. Сжатие поддерживается только для DIB с цветовым разрешением 4 или 8 бит/пиксел. DIB-секцию сжать нельзя.

Параметр	Описание
pDC	Указатель на контекст дисплея
bCompress	*TRUE* (по умолчанию) означает сжатие DIB, а *FALSE* — развертывание
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***CopyToMapFile*** создает новый проецируемый в память файл и копирует в него данные из существующего DIB. При этом освобождается вся выделенная ранее память, и закрывается прежний проецируемый файл (если таковой был). Дан-

ные не записываются на диск, пока новый файл не закрывается, что происходит при повторном использовании или уничтожении объекта *CDib*.

Параметр	Описание
strPathname	Полное имя проецируемого файла
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***CreateBitmap*** создает из существующего DIB-изображения GDI-растр. Вызывается функцией *Compress*. Не путайте эту функцию с *CreateSection*, которая создает DIB и запоминает описатель.

Параметр	Описание
pDC	Указатель на контекст дисплея или принтера
Возвращаемое значение	Описатель растрового изображения GDI (при неудаче — NULL); *не* записывается в открытую переменную-член

- ***CreateSection*** создает DIB-секцию, вызывая функцию Win32 *CreateDIBSection*. Память, выделенная под изображение, не инициализируется.

Параметр	Описание
PDC	Указатель на контекст дисплея или принтера
Возвращаемое значение	Описатель GDI-растра (при неудаче — NULL); также записывается в открытую переменную-член

- ***Draw*** выводит объект *CDib* на дисплей (или принтер), вызывая функцию Win32 *StretchDIBits*. При необходимости растровое изображение растягивается для подгонки под заданный прямоугольник.

Параметр	Описание
PDC	Указатель на контекст дисплея или принтера — получатель DIB-изображения
Origin	Объект *CPoint*, содержащий логические координаты, в которые выводится DIB
Size	Объект *CSize*, в логических единицах задающий размеры прямоугольника на устройстве вывода
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***Empty*** очищает DIB, освобождая при необходимости выделенную память и закрывая спроецированный в память файл.
- ***GetDimensions*** возвращает ширину и высоту DIB в пикселах.

Параметр	Описание
Возвращаемое значение	Объект *Csize*

- ***GetSizeHeader*** возвращает общее число байт в информационном заголовке и цветовой таблице.

Параметр	Описание
Возвращаемое значение	32-разрядное целое

- ***GetSizeImage*** возвращает размер изображения DIB в байтах (исключая информационный заголовок и цветовую таблицу).

Параметр	Описание
Возвращаемое значение	32-разрядное целое

- ***MakePalette*** считывает цветовую таблицу и, если эта таблица существует, создает палитру Windows. Описатель *HPALETTE* запоминается в переменной-члене.

Параметр	Описание
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***Read*** считывает DIB из файла в объект *CDib*. Файл должен быть предварительно успешно открыт. Если файл в формате BMP, чтение осуществляется с начала файла. Если же файл представляет собой документ, чтение начинается с текущей позиции указателя файла.

Параметр	Описание
PFile	Указатель на объект *CFile*; соответствующий файл на диске содержит DIB
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***ReadSection*** считывает заголовок из BMP-файла, вызывает *CreateDIBSection* для выделения памяти, а затем считывает в эту память изображение из файла. Эта функция полезна, если нужно считать DIB-изображение с диска и отредактировать его вызовами GDI. Отредактированное изображение можно записать на диск, используя функции *Write* или *CopyToMapFile*.

Параметр	Описание
PFile	Указатель на объект *CFile*; соответствующий файл на диске содержит DIB
PDC	Указатель на контекст устройства дисплея или принтера
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***Serialize*** выполняет сериализацию (см. главу 16). Функция *CDib::Serialize*, переопределяющая MFC-функцию *CObject::Serialize*, вызывает функции-члены *Read* и *Write*. Параметры описаны в *Microsoft Foundation Class Library Reference*.
- ***SetSystemPalette*** вызывается для DIB-изображения с цветовым разрешением 16, 24 или 32 бит/пиксел, у которого нет цветовой таблицы, чтобы создать для объекта *CDib* логическую палитру, соответствующую палитре, возвращаемой функцией *CreateHalftonePalette*. Если ваша программа работает с 256-цветным дисплеем, поддерживающим палитры, и вы не вызываете *SetSystemPalette*, то при выводе DIB используются только 20 стандартных цветов Windows (так как палитра не задана).

Параметр	Описание
pDC	Указатель на контекст дисплея
Возвращаемое значение	*TRUE* — при благополучном завершении

- ***UsePalette*** подключает логическую палитру объекта *CDib* к контексту устройства и реализует ее. Функция-член *Draw* вызывает *UsePalette* перед рисованием DIB.

Параметр	Описание
pDC	Указатель на контекст устройства дисплея (для реализации палитры).
bBackground	Если этот флаг равен *FALSE* (по умолчанию) и приложение выполняется на переднем плане, Windows реализует эту палитру как палитру переднего плана (копирует в системную палитру максимально возможное число цветов). В противном случае (*TRUE*) Windows реализует палитру как фоновую (стремясь наилучшим образом отобразить логическую палитру на системную).
Возвращаемое значение	Число элементов логической палитры, отображенных на системную палитру. В случае ошибки возвращается значение *GDI_ERROR*.

- ***Write*** записывает DIB из объекта *CDib* в файл. Файл должен быть предварительно успешно открыт или создан.

Параметр	Описание
PFile	Указатель на объект *CFile*; DIB записывается в соответствующий дисковый файл
Возвращаемое значение	*TRUE* — при благополучном завершении

Для удобства четыре открытых переменных-члена обеспечивают доступ к памяти DIB и к описателю DIB-секции. Эти переменные дают ключ к структуре объекта *CDib*. Он представляет собой набор указателей на память в куче. Владельцем этой памяти может быть и DIB. Дополнительные закрытые переменные-члены определяют, должен ли класс *CDib* освобождать память.

Производительность при выводе DIB-изображения на дисплей

Оптимизированная обработка DIB — одна из главных особенностей Windows. У современных видеокарт есть специальные буферы, поддерживающие стандартный формат DIB-изображений. Если программа выполняется на компьютере с такой платой, она может задействовать преимущества нового *DIB-процессора* (DIB engine) Windows, который ускоряет процесс прямого рисования изображений из DIB. Если же программа выполняется в режиме VGA, вам не повезло: она будет работать медленнее, хотя и вполне корректно.

При работе с Windows в режиме 256 цветов изображения 8 бит/пиксел будут выводиться на дисплей очень быстро как при помощи *StretchBlt*, так и *StretchDIBits*. Но при выводе растровых изображений 16 или 24 бит/пиксел эти функции работают слишком медленно. Добиться ускорения вывода можно, создав отдельный GDI-растр с 8 бит/пиксел с последующим вызовом *StretchBlt*. Конечно, при этом надо реализовать корректную палитру перед созданием изображения и выводом его на экран.

Следующий фрагмент кода можно вставить сразу после загрузки объекта *CDib* из BMP-файла:

```
// m_hBitmap - переменная-член типа HBITMAP
// m_dcMem - объект контекста устройства в памяти класса CDC
m_pDib->UsePalette(&dc);
m_hBitmap = m_pDib->CreateBitmap(&dc);     // может работать медленно
::SelectObject(m_dcMem.GetSafeHdc(), m_hBitmap);
```

А этот код можно использовать вместо *CDib::Draw* в функции *OnDraw* класса «вид»:

```
m_pDib->UsePalette(&dc);  // можно поместить в обработчик сообщения палитры
CSize sizeDib = m_pDib->GetDimensions();
pDC->StretchBlt(0, 0, sizeDib.cx, sizeDib.cy, &m_dcMem,
                0, 0, sizeToDraw.cx, sizeToDraw.cy, SRCCOPY);
```

Не забудьте вызвать *DeleteObject* для *m_hBitmap*, когда закончите работу с ним.

Пример Ex06d

Теперь посмотрим, как *CDib*-класс работает в приложении. Ex06d воспроизводит два DIB-изображения: одно из ресурса, а другое из BMP-файла. Программа управляет системной палитрой и корректно выводит DIB на принтер.

Давайте обсудим процесс создания программы Ex06d. Неплохо было бы вручную ввести код класса «вид», но лучше взять готовые файлы cdib.h и cdib.cpp с компакт-диска.

1. **С помощью MFC Application Wizard создайте проект Ex06d.** Примите параметры, предлагаемые по умолчанию, но выберите Single Document и базовый класс *CScrollView* для класса *CEx06dView*.
2. **Импортируйте растровое изображение *Red Blocks*.** В меню Project выберите команду Add Resource. В диалогом окне Add Resource щелкните кнопку Import. Импортируйте растровое изображение Red Blocks.bmp из каталога \vcppnet\bitmaps на компакт-диске. Visual C++ .NET скопирует этот файл в подкаталог \res вашего проекта. Назначьте изображению идентификатор *IDB_REDBLOCKS* и сохраните внесенные изменения.
3. **Добавьте в проект класс *CDib*.** Если проект создавался с нуля, скопируйте файлы cdib.h и cdib.cpp из каталога \vcppnet\ex06d на компакт-диске. Но простого копирования недостаточно — нужно добавить эти файлы в проект: выберите команду Add Existing Item из меню Project среды разработки, а затем в списке выберите файлы cdib.h и cdib.cpp и щелкните кнопку OK. Теперь в окне ClassView или Solution Explorer вы увидите класс *CDib* со всеми его членами — переменными и функциями.
4. **Добавьте закрытые переменные-члены типа *CDib* в класс *CEx06dView*.** В конце заголовочного файла CEx06dView.h вставьте строки:

```
private:
    CDib m_dibFile;
    CDib m_dibResource;
```

   А в начало файла Ex06dView.h:

```
#include "cdib.h"
```

5. **Отредактируйте функцию *OnInitialUpdate* в файле Ex06dView.cpp.** Эта функция устанавливает режим преобразования координат в *MM_HIMETRIC* и загружает объект *m_dibResource* из ресурса *IDB_REDBLOCKS*. Объект к ресурсу в EXE-файле подсоединяет функция *CDib::AttachMemory*. Введите выделенный код:

```
void Cex06dView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize totalSize(30000, 40000);   // 30x40 см
    CSize lineSize = CSize(totalSize.cx / 100, totalSize.cy / 100);
    SetScrollSizes(MM_HIMETRIC, totalSize, totalSize, lineSize);
    LPVOID lpvResource = (LPVOID) ::LoadResource(NULL,
        ::FindResource(NULL, MAKEINTRESOURCE(IDB_REDBLOCKS), RT_BITMAP));
    m_dibResource.AttachMemory(lpvResource);   // ::LockResource не требуется
    CClientDC dc(this);
    TRACE("bits per pixel = %d\n", dc.GetDeviceCaps(BITSPIXEL));
}
```

6. **Отредактируйте функцию-член *OnDraw* в файле Ex06dView.cpp.** В коде этой функции вызывается *CDib::Draw* для каждого из двух DIB. *UsePalette* должны вызывать на самом деле обработчики сообщений *WM_QUERYNEWPALETTE* и *WM_PALETTECHANGED*. С этими сообщениями работать довольно трудно, так как их не посылают непосредственно объекту «вид», поэтому мы прибегнем к упрощению. Добавьте выделенный код:

```
void CEx06dView::OnDraw(CDC* pDC)
{
    BeginWaitCursor();
    m_dibResource.UsePalette(pDC);   // это должно быть не здесь,
    m_dibFile.UsePalette(pDC);       // а в обработчиках сообщений палитры
    pDC->TextOut(0, 0, "Press the left mouse button here to load a file.");
    CSize sizeResourceDib = m_dibResource.GetDimensions();
    sizeResourceDib.cx *= 30;
    sizeResourceDib.cy *= -30;
    m_dibResource.Draw(pDC, CPoint(0, -800), sizeResourceDib);
    CSize sizeFileDib = m_dibFile.GetDimensions();
    sizeFileDib.cx *= 30;
    sizeFileDib.cy *= -30;
    m_dibFile.Draw(pDC, CPoint(1800, -800), sizeFileDib);
    EndWaitCursor();
}
```

7. **Создайте в классе *CEx06dView* обработчик сообщения *WM_LBUTTONDOWN*.** Отредактируйте файл Ex06dView.cpp. *OnLButtonDown* содержит код для считывания DIB двумя способами. Если вы оставите определение *MEMORY_MAPPED_FILES*, активизируется код, использующий *AttachMapFile* для считывания проецируемого в память файла. Если же вы закомментируете первую строку, активизируется вызов *Read*. Вызов *SetSystemPalette* присутствует здесь специально для DIB, у которых нет цветовой таблицы. Введите выделенный код:

```
#define MEMORY_MAPPED_FILES
void CEx10cView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CFileDialog dlg(TRUE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) {
        return;
    }
#ifdef MEMORY_MAPPED_FILES
    if (m_dibFile.AttachMapFile(dlg.GetPathName(),
            TRUE) == TRUE) { // совместное использование
        Invalidate();
    }
#else
    CFile file;
    file.Open(dlg.GetPathName(), CFile::modeRead);
    if (m_dibFile.Read(&file) == TRUE) {
        Invalidate();
    }
#endif // MEMORY_MAPPED_FILES
    CClientDC dc(this);
    m_dibFile.SetSystemPalette(&dc);
}
```

8. **Соберите и запустите приложение.** В каталоге проекта Ex06d на компакт-диске есть несколько интересных растровых изображений. Файл chicago.bmp — это DIB с 8 бит/пиксел и цветовой таблицей с 256 элементами; forest.bmp и clouds.bmp тоже имеют цветовое разрешение 8 бит/пиксел, но их цветовые таблицы меньше; balloons.bmp — это DIB с 24 бит/пиксел без цветовой таблицы. Попробуйте и другие BMP-файлы, если они у вас есть. Кстати, Red Blocks — это 16-цветное DIB-изображение, в котором используются стандартные цвета, всегда имеющиеся в системной палитре.

Еще несколько слов о DIB

Каждая новая версия Windows предлагает новые возможности программирования DIB. Windows 2000 поддерживает функции *LoadImage* и *DrawDibDraw*, альтернативные уже описанным нами DIB-функциям. Поэкспериментируйте с этими функциями и посмотрите, как они работают в ваших приложениях.

Функция *LoadImage*

Функция *LoadImage* может считывать растровое изображение прямо из дискового файла, возвращая описатель DIB-секции. Допустим, нам нужно добавить функцию-член к классу *CDib*, которая действовала бы, как *ReadSection*. Вот какой код можно добавить в cdib.cpp:

```
BOOL CDib::ImageLoad(const char* lpszPathName, CDC* pDC)
{
    Empty();
    m_hBitmap = (HBITMAP) ::LoadImage(NULL, lpszPathName, IMAGE_BITMAP, 0, 0,
```

```
        LR_LOADFROMFILE | LR_CREATEDIBSECTION | LR_DEFAULTSIZE);
    DIBSECTION ds;
    VERIFY(::GetObject(m_hBitmap, sizeof(ds), &ds) == sizeof(ds));
    // Выделить память для BITMAPINFOHEADER
    // и наибольшей возможной таблицы цветов
    m_lpBMIH = (LPBITMAPINFOHEADER) new char[sizeof(BITMAPINFOHEADER) +
        256 * sizeof(RGBQUAD)];
    memcpy(m_lpBMIH, &ds.dsBmih, sizeof(BITMAPINFOHEADER));
    TRACE("CDib::LoadImage, biClrUsed = %d, biClrImportant = %d\n",
        m_lpBMIH->biClrUsed, m_lpBMIH->biClrImportant);
    ComputeMetrics(); // устанавливает m_lpvColorTable
    m_nBmihAlloc = crtAlloc;
    m_lpImage = (LPBYTE) ds.dsBm.bmBits;
    m_nImageAlloc = noAlloc;
    // Получить таблицу цветов DIB секции
    // и создать из нее палитру
    CDC memdc;
    memdc.CreateCompatibleDC(pDC);
    ::SelectObject(memdc.GetSafeHdc(), m_hBitmap);
    UINT nColors = ::GetDIBColorTable(memdc.GetSafeHdc(), 0, 256,
        (RGBQUAD*) m_lpvColorTable);
    if (nColors != 0) {
        ComputePaletteSize(m_lpBMIH->biBitCount);
        MakePalette();
    }
    // контекст устройства в памяти уничтожается,
    // растровое изображение отключается
    return TRUE;
}
```

Эта функция извлекает и копирует структуру *BITMAPINFOHEADER* и устанавливает значения указателей-членов класса *CDib*. Чтобы извлечь палитру из DIB-секции, нужно потрудиться, но Win32-функция *GetDIBColorTable* послужит в этом хорошим подспорьем. Интересно, что *GetDIBColorTable* не может сообщить, сколько элементов палитры использует данное DIB-изображение. Если, к примеру, в DIB применяются только 60 элементов, *GetDIBColorTable* строит цветовую таблицу из 256 элементов со 196 элементами, установленными в 0.

Функция *DrawDibDraw*

Windows включает компонент Video for Windows (VFW), поддерживаемый Visual C++ .NET. VFW-функция *DrawDibDraw* — альтернатива функции *StretchDIBits*. Одно из преимуществ *DrawDibDraw* — возможность использовать *смешанные* (dithered) цвета, другое преимущество — ускоренное рисование изображений с числом битов на пиксел, не совпадающим с текущим видеорежимом. Основной же недостаток — необходимость подключать код VFW к прикладному процессу во время исполнения.

Ниже приведена функция *DrawDib* класса *CDib*, которая вызывает *DrawDibDraw*:

```
BOOL CDib::DrawDib(CDC* pDC, CPoint origin, CSize size)
{
   if (m_lpBMIH == NULL) return FALSE;
   if (m_hPalette != NULL) {
      ::SelectPalette(pDC->GetSafeHdc(), m_hPalette, TRUE);
   }
   HDRAWDIB hdd = ::DrawDibOpen();
   CRect rect(origin, size);
   pDC->LPtoDP(rect);      // Преобразуем координаты прямоугольника
                           // DIB-изображения в режим MM_TEXT
   rect -= pDC->GetViewportOrg();
   int nMapModeOld = pDC->SetMapMode(MM_TEXT);
   ::DrawDibDraw(hdd, pDC->GetSafeHdc(), rect.left, rect.top,
      rect.Width(), rect.Height(), m_lpBMIH, m_lpImage, 0, 0,
      m_lpBMIH->biWidth, m_lpBMIH->biHeight, 0);
   pDC->SetMapMode(nMapModeOld);
   VERIFY(::DrawDibClose(hdd));
   return TRUE;
}
```

DrawDibDraw требует координат *MM_TEXT* и режима преобразования координат *MM_TEXT*. Таким образом, логические координаты следует преобразовать не в координаты устройства, а в пикселы — с начальной точкой в левом верхнем углу.

Чтобы использовать *DrawDibDraw*, в программу надо включить оператор *#include <vfw.h>*, а также подключить библиотеку vfw32.lib в список входных файлов загрузчика. *DrawDibDraw* полагает, что выводимое ею изображение располагается в памяти с доступом для чтения и записи, — помните об этом, выделяя память под BMP-файл.

Растровые изображения на командных кнопках

Библиотека MFC облегчает вывод на командных кнопках растровых изображений вместо текста. При программировании с нуля следовало бы установить для кнопки стиль Owner Draw и написать обработчик сообщения в классе диалогового окна, который рисовал бы растровое изображение в окне элемента управления «кнопка». А при использовании MFC-класса *CBitmapButton* задача заметно упрощается, хотя надо придерживаться строго определенной процедуры. Не очень переживайте насчет того, как все это работает, а радуйтесь, что не придется писать много кода.

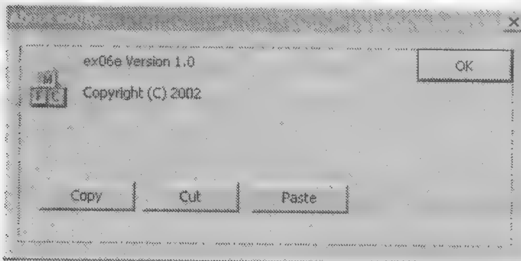
Короче, вы создаете, как обычно, диалоговый ресурс, закрепляя за кнопками, которые будут отображать растровые изображения, уникальные текстовые заголовки. Затем добавляете к проекту несколько ресурсов растровых изображений, но идентифицируете их по *именам*, а не числовым идентификаторам. Наконец, дополняете класс диалогового окна несколькими переменными-членами типа *CBitmapButton* и вызываете для каждого из них функцию-член *AutoLoad*, которая сопоставляет имя растрового изображения и текст кнопки. Если текст на кнопке — «COPY», вы добавляете два растра: «COPYU» (кнопка отжата) и «COPYD» (кнопка нажата). Да, кстати, не забудьте установить стиль кнопок Owner Draw. Все станет понятнее, когда вы сами напишете программу.

Примечание Взглянув на код MFC для класса *CBitmapButton*, можно заметить, что это растровое изображение — обычный GDI-растр, отображаемый с помощью *BitBlt*. Поэтому палитра здесь не поддерживается. Но обычно это не проблема, так как растровые изображения для кнопок чаще всего 16-цветные; в них используются стандартные цвета VGA.

Пример Ex06e

Этот пример демонстрирует вывод растровых изображений на командных кнопках.

1. **С помощью MFC Application Wizard создайте проект Ex06e.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview.
2. **Модифицируйте диалоговый ресурс *IDD_ABOUTBOX* в окне Resource View.** Чтобы не создавать новое диалоговое окно из-за одних кнопок, возьмем диалоговое окно About, которое MFC Application Wizard автоматически генерирует для каждого проекта. Добавьте три командные кнопки (см. рисунок) с указанными названиями, оставив идентификаторы кнопок по умолчанию: *IDC_BUTTON1*, *IDC_BUTTON2* и *IDC_BUTTON3*. Размеры кнопок значения не имеют — в период выполнения программы MFC подгоняет их под размеры растровых изображений.



Установите в TRUE свойство *Owner Draw* всех трех кнопок.

3. **Импортируйте три растровых изображения (EditCopy.bmp, EditPast.bmp и EditCut.bmp) из подкаталога \vcppnet\Ex06e на компакт-диске.** В меню Project выберите команду Add Resource и в открывшемся окне щелкните кнопку Import. Начните с EditCopy.bmp. Назначьте ей имя «COPYU».

   Обязательно заключите имя в кавычки, чтобы идентифицировать ресурс по имени, а не по номеру. Теперь у нас есть растровое изображение для кнопки в «отжатом» состоянии. Закройте окно растрового изображения и скопируйте из окна Resource View растровое изображение через буфер обмена или перетащите его. Переименуйте копию в «COPYD» (кнопка в «нажатом» состоянии) и отредактируйте ее. Выберите из меню Image команду Invert Colors. Создать изображение отжатой кнопки можно и иначе, но инверсия — самый быстрый способ.

   Повторите эти операции для растровых изображений EditCut и EditPast. В результате вы должны получить в своем проекте такие ресурсы:

Имя ресурса	Исходный файл	Инвертированные цвета
«COPYU»	EditCopy.bmp	Нет
«COPYD»	EditCopy.bmp	Да
«CUTU»	EditCut.bmp	Нет
«CUTD»	EditCut.bmp	Да
«PASTEU»	EditPast.bmp	Нет
«PASTED»	EditPast.bmp	Да

4. **Отредактируйте код класса *CAboutDlg*.** И объявление, и реализацию этого класса содержит файл Ex06e.cpp. Для начала добавьте в объявление класса три закрытых переменных-члена:

```
private:
   CBitmapButton m_editCopy;
   CBitmapButton m_editCut;
   CBitmapButton m_editPaste;
```

   С помощью мастера окна Properties утилиты Class View переопределите виртуальную функцию *OnInitDialog*. (Убедитесь, что при этом выбран класс *CAboutDlg*.) Код обработчика сообщения выглядит так:

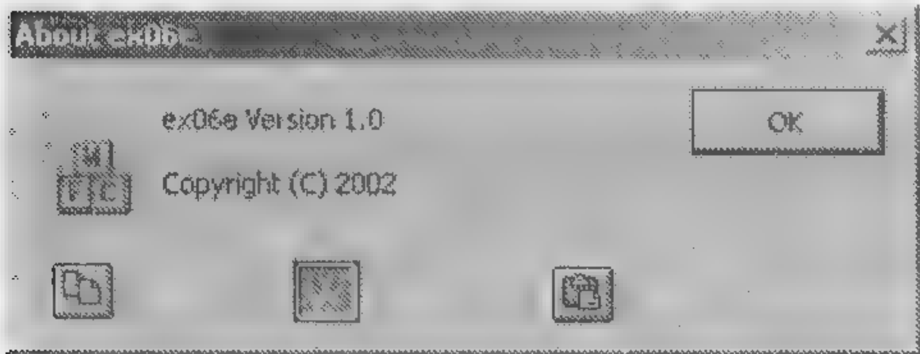
```
BOOL CAboutDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   VERIFY(m_editCopy.AutoLoad(IDC_BUTTON1, this));
   VERIFY(m_editCut.AutoLoad(IDC_BUTTON2, this));
   VERIFY(m_editPaste.AutoLoad(IDC_BUTTON3, this));
   return TRUE; // return TRUE unless you set the focus to a control,
   // EXCEPTION: OCX Property Pages should return FALSE
}
```

   Функция *AutoLoad* устанавливает связь между кнопкой и двумя соответствующими ресурсами. *VERIFY* — это диагностический MFC-макрос, выводящий информационное окно, если имена растровых изображений заданы неверно.

5. **Отредактируйте функцию *OnDraw* в файле Ex06eView.cpp.** Замените код, созданный MFC Application Wizard, следующей строкой (не забудьте раскомментировать определение переменной *pDC*):

```
pDC->TextOut(0, 0, "Choose About from the Help menu.");
```

6. **Соберите и запустите приложение.** После запуска программы выберите из меню Help команду About и понаблюдайте за поведением кнопок. На рисунке кнопка CUT изображена в «нажатом» состоянии.



Кнопки с растровыми изображениями, как и обычные кнопки, отправляют уведомляющие сообщения *BN_CLICKED*. Естественно, мастера Class View позволяют создать в классе диалогового окна обработчики этих сообщений.

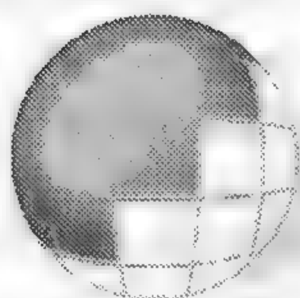
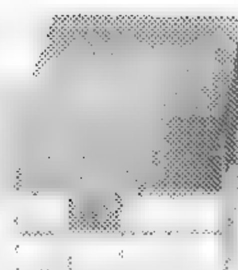
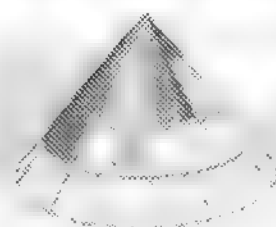
И еще пара слов о растровых изображениях на кнопках

Мы использовали растровые изображения для отображения «нажатых» и «отжатых» кнопок. Класс *CBitmapButton* поддерживает, кроме того, растровые изображения для кнопок, имеющих фокус ввода, и для кнопок в *отключенном* (disabled) состоянии. Для кнопки Copy имя растрового изображения в фокусе было бы «COPYF», а имя «вне фокуса» — «COPYX». Чтобы протестировать работу отключенной кнопки, создайте растровое изображение «COPYX», например, перечеркнутое красной линией, и вставьте в программу строку:

```
m_editCopy.EnableWindow(FALSE);
```

ГЛАВА

7

Диалоговые окна

Практически все Windows-программы взаимодействуют с пользователем через диалоговые окна. Диалоговое окно может просто содержать сообщение и кнопку OK, а может быть и очень сложной формой для ввода данных. Диалоговое окно можно перемещать и закрывать; оно принимает сообщения и даже отрабатывает команды отрисовки данных в своей клиентской области.

Существует два типа диалоговых окон: *модальные* (modal) и *немодальные* (modeless). В этой главе мы познакомимся с обоими. Мы также поговорим о стандартных диалоговых окнах Windows, предназначенных для открытия файлов, выбора шрифтов и пр.

Модальные и немодальные диалоговые окна

Базовый класс *CDialog* поддерживает как модальные, так и немодальные диалоговые окна. Пока открыто модальное диалоговое окно (например, Open File — «Открытие файла»), пользователю недоступны другие окна программы (точнее, окна того же потока пользовательского интерфейса). А немодальное диалоговое окно позволяет работать с другими окнами программы. Пример — диалоговое окно Find and Replace редактора Microsoft Word, которое совершенно не мешает редактировать текст.

Выбор конкретного типа диалогового окна определяется характером создаваемого приложения. Программировать модальные диалоговые окна намного проще, и это часто влияет на решение программиста.

Ресурсы и элементы управления

Итак, диалоговое окно — это настоящее, полноценное окно. Но чем же оно отличается от окон класса *CView*, с которыми вы успели поработать? Хотя бы тем, что

диалоговое окно почти всегда связано с каким-нибудь ресурсом Windows, идентифицирующим элементы и определяющим структуру окна. Поскольку диалоговый ресурс можно создавать и модифицировать в редакторе диалоговых окон (одном из редакторов ресурсов), диалоговые окна формируются быстро, эффективно и наглядно.

Диалоговое окно содержит набор *элементов управления* (controls): *поля ввода* (edit control; их еще называют *текстовыми окнами* — text box), *кнопки* (button), *списки* (list box), *поля со списками* (combo box), *статический текст* (static text) или *метки* (labels), *древовидные списки* (tree views), *индикаторы хода процесса* (progress indicators), *ползунки* (sliders) и др. Windows управляет этими элементами, используя специальную, значительно упрощающую труд программиста логику группировки и обхода. На элементы управления ссылаются по указателю на *CWnd* (так как элементы сами являются окнами) либо по индексу (с сопоставленной средствами *#define* константой) назначенному в ресурсе. В ответ на действия пользователя, скажем, ввод текста или щелчок кнопки, элементы управления передают сообщения родительскому диалоговому окну.

Тесное взаимодействие библиотеки MFC и Microsoft Visual Studio заметно облегчает программирование диалоговых окон Windows. Visual Studio генерирует класс, производный от *CDialog*, а затем позволяет сопоставить переменные-члены класса «диалоговое окно» элементам управления. Вы можете указывать такие параметры, как максимальная длина текста или границы диапазона вводимых чисел. После этого Visual Studio генерирует вызовы MFC-функций, отвечающих за обмен данными и проверку их корректности. Эти функции перемещают данные между экранными элементами и переменными-членами соответствующего класса.

Программирование модального диалогового окна

Модальные диалоговые окна встречаются чаще. Пользователь что-то делает (скажем, выбирает команду в меню), на экране появляется диалоговое окно, пользователь вводит данные, а затем закрывает его. Чтобы дополнить существующий проект модальным диалоговым окном, сделайте так.

1. Используя редактор диалоговых окон, создайте диалоговый ресурс с элементами управления. Редактор обновит файл описания ресурсов (RC-файл) вашего проекта, включив в него новый ресурс, и файл resource.h, дополнив его соответствующими константами *#define*.
2. С помощью MFC Class Wizard создайте класс «диалоговое окно», производный от *CDialog*, и закрепите его за ресурсом, созданным на шаге 1. Visual Studio добавит в проект требуемый код и заголовочный файл.

Примечание При генерации производного класса диалогового окна Visual Studio формирует конструктор, который запускает конструктор *CDialog*, принимающий в качестве параметра идентификатор ресурса. Заголовочный файл диалогового окна содержит константу класса *IDD*, которой присвоен идентификатор диалогового ресурса. А реализация конструктора в CPP-файле выглядит так:

```
IMPLEMENT_DYNAMIC (CMyDialog, CDialog)
CMyDialog::CMyDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMyDialog::IDD, pParent)
{
    // здесь должен быть инициализирующий код
}
```

Применение *enum IDD* избавляет CPP-файл от идентификаторов ресурсов, определяемых в файле resource.h данного проекта[1].

3. Добавьте в класс диалогового окна переменные-члены и функции, предназначенные для обмена и проверки данных.
4. В окне Properties утилиты Class View добавьте обработчики сообщений для кнопок и других (генерирующих события) элементов управления диалогового окна.
5. Напишите код для инициализации (в *OnInitDialog*) элементов управления и для обработчиков сообщений. Убедитесь, что при закрытии диалогового окна (если только пользователь не нажмет кнопку отмены) вызывается виртуальная функция-член *OnOK* класса *CDialog* (она вызывается по умолчанию).
6. В своем классе «вид» напишите код, активизирующий диалоговое окно. Он сводится к вызову конструктора вашего класса диалогового окна и последующему вызову функции-члена *DoModal* класса диалогового окна. *DoModal* возвращает управление только после закрытия диалогового окна.

   А теперь рассмотрим все эти операции на примере.

Пример Ex07a: Диалоговое окно «каждой твари по паре»

Мы не станем возиться с каким-нибудь простеньким примером, а встроим в наше диалоговое окно почти все возможные элементы управления. Это несложно — ведь нам поможет редактор диалоговых окон Visual Studio. Готовое диалоговое окно показано на рис. 7-1.

Как видите, это диалоговое окно предназначено для учета кадров. Это пример довольно скучного бизнес-приложения — программу слегка оживляют полосы прокрутки «loyalty» («преданность») и «reliability» («надежность») — классический пример прямого ввода и наглядного отображения данных. Еще интереснее были бы здесь элементы управления ActiveX, но использовать их вы научитесь только в главе 9.

Построение диалогового ресурса

Давайте создадим диалоговый ресурс.

1. **Запустите MFC Application Wizard и создайте MFC-проект Ex07a.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex07a.

[1] Файл resource.h все равно приходится использовать в CPP-файле — для доступа к дочерним элементам управления по их идентификаторам. — *Прим. перев.*

Рис. 7-1. *Готовое диалоговое окно в действии*

На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.

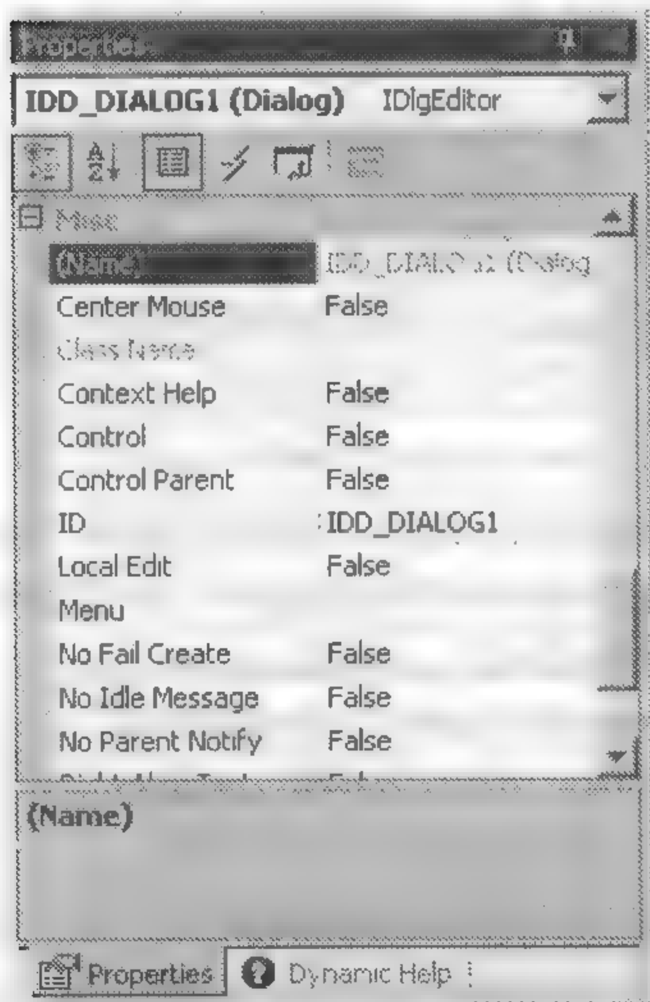
2. **Создайте новый диалоговый ресурс с идентификатором *IDD_DIALOG1*.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс.

Редактор диалоговых окон

Редактор диалоговых окон присвоит новому диалоговому окну идентификатор ресурса *IDD_DIALOG1*. Заметьте: он вставляет в новое диалоговое окно кнопки OK и Cancel.

3. **Измените размер диалогового окна и присвойте ему заголовок.** Увеличьте размеры окна редактирования, чтобы было удобно работать.

   Щелкните новое диалоговое окно правой кнопкой и выберите в контекстном меню команду Properties — откроется одноименное окно (обычно оно располагается в правом нижнем углу).

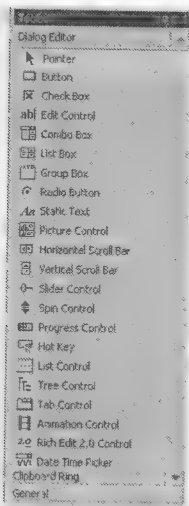


   Значок кнопки в заголовке окна Properties определяет ее видимость — при нажатой кнопке оно располагается поверх остальных окон. В окне Properties измените значение свойства Caption на «The Dialog Box That Ate Cincinnati»[1], а свойства System Menu — на FALSE.

4. **Разместите элементы управления в диалоговом окне.** Это делается в окне Toolbox. (Если его не видно, выберите в меню View команду Toolbox.) Перетащите нужные элементы в создаваемое диалоговое окно, а затем сдвиньте и измените их размер (рис. 7-1). Окно Toolbox показано ниже.

Примечание Редактор диалоговых окон сообщает о положении и размере каждого элемента управления в правой части строки состояния. При этом координаты выражаются *не* в аппаратных, а в специальных «единицах диалога» (dialog unit, DLU). Горизонтальная DLU — это средняя ширина шрифта, используемого в диалоговом окне, деленная на 4; вертикальная DLU — средняя высота этого шрифта, деленная на 8. Ну, а шрифт — это обычно MS Sans Serif размером 8 пт.

[1] В свободном переводе на русский эту англоязычную идиому с аллюзией на фильм «Cockroach that ate Cincinnati» («Таракан, сожравший город Цинциннати»), можно перевести как «диалоговое окно, охватывающее все возможные элементы управления». Мы предпочли перевести название приложения как «Диалоговое окно «каждой твари по паре» — *Прим. перев.*



Теперь коротко рассмотрим элементы управления нашего диалогового окна.

- **Статический текст для поля Name (имя).** Этот элемент просто выводит на экран указанные символы и собственно к диалогу с пользователем отношения не имеет. Расположив ограничивающий прямоугольник, введите текст (при этом изменится свойство Caption в окне Properties); размеры прямоугольника можно изменить. Создайте текстовое поле Name и присвойте его свойству Caption значение &Name. Другие статические элементы создавайте по аналогии. У них у всех один и тот же идентификатор, но это не важно, поскольку доступ к ним программе не нужен.

Примечание Если в свойстве Caption статического текста есть знак амперсанд (&), то во время исполнения следующий за ним символ подчеркивается. Это значит, что при нажатии клавиши Alt и соответствующего символа пользователь получает моментальный доступ к относящемуся к статическому тексту элементу управления. Причем этот элемент управления должен идти в последовательности переходов (tabbing order) сразу за статическим текстом. (Мы обсудим последовательность переходов чуть позже.) Таким образом, нажатие Alt+N позволяет «перескакивать» в текстовое поле Name, а Alt+K — в поле Skill (рис. 7-1). Думаю, не стоит говорить, что подчеркнутые символы должны быть уникальными в рамках диалогового окна. Именно поэтому в элементе управления Skill используется символ K — ведь S зарезервирован для поля SS Nbr.

- **Поле ввода Name.** Поле ввода — основное средство ввода текста в диалоговых окнах. Смените его идентификатор с *IDC_EDIT1* на *IDC_NAME*. Остальным свойствам оставьте значения по умолчанию. Заметьте: Auto HScroll по умолчанию устанавливается в TRUE, т. е. текст по мере заполнения текстового поля прокручивается справа налево.
- **Поле ввода SS Nbr (social security number – номер карточки социального страхования).** Этот элемент управления идентичен предыдуще-

му. Замените его идентификатор на *IDC_SSN*. Впоследствии при помощи мастера Add Member Variable Wizard вы превратите это поле в числовое.

Примечание Чтобы выровнять несколько элементов управления, выделите их и выберите команду выравнивания (Lefts, Centers, Rights, Tops, Middles или Bottoms) из подменю Align меню Format.

Вы также можете выровнять элементы управления по сетке. Чтобы включить сетку, щелкните кнопку Toggle Grid (на панели инструментов Dialog Editor).

- **Поле ввода Bio (biography — биография).** Это многострочное поле ввода. Чтобы оно стало таковым, установите свойство Multiline в TRUE. Присвойте ему идентификатор *IDC_BIO*.
- **Группирующая рамка Category (категория).** Нужна только для визуальной группировки двух переключателей. Введите ее название: *&Category*. Идентификатор, присвоенный по умолчанию, нас устроит.
- **Переключатели Hourly (почасовая оплата) и Salary (оклад).** Разместите эти переключатели в группирующей рамке Category. Свойство Caption переключателя Hourly задайте как *IDC_CAT*, а Group и Tabstop установите в TRUE. У переключателя Salary определите свойства: Caption — в Salary, а Tabstop в TRUE.

  Убедитесь, что у обоих переключателей свойство Auto установлено в TRUE (по умолчанию) и что только у кнопки Hourly установлено свойство Group. Последнее означает, что Hourly — первый переключатель в группе Category. При правильном задании этих свойств Windows гарантирует, что в установленном положении будет только один переключатель. Группирующая рамка Category на их функционирование не влияет.
- **Группирующая рамка Insurance (страховка).** В этом элементе управления размещаются три флажка. Введите название рамки — *&Insurance*.

Примечание Позже, установив порядок обхода элементов в диалоговом окне, вы добьетесь, чтобы группирующая рамка Insurance следовала за последним переключателем рамки Category. Только не забудьте установить в TRUE свойство Group элемента управления Insurance, чтобы «завершить» предыдущую группу. Если этого не сделать, особой беды не будет, но, запустив программу под отладчиком, вы получите пару-тройку предупреждений.

- **Флажки Life (жизни), Disability (на случай инвалидности) и Medical (медицинская).** Разместите эти элементы управления в группирующей рамке Insurance. Примите свойства, предлагаемые по умолчанию, но замените идентификаторы на *IDC_LIFE*, *IDC_DIS* и *IDC_MED*. В отличие от переключателей флажки ведут себя независимо — пользователь сможет устанавливать любую их комбинацию.
- **Поле со списком Skill (профессиональные навыки).** Это первый из трех типов полей со списком. Изменив идентификатор на *IDC_SKILL*, установи-

те свойство Type в Simple. Растяните элемент управления в высоту, чтобы на нем разместилось несколько строк. Теперь щелкните свойство Data и введите, разделяя их точкой с запятой, названия трех профессий: Manager (менеджер), Programmer (программист) и Writer («писатель»).

Этот тип поля со списком называется *простым* (Simple). В верхнем поле ввода пользователь может вводить любые данные и при помощи мыши или клавиш-стрелок «вверх» или «вниз» выделять элемент в окне списка.

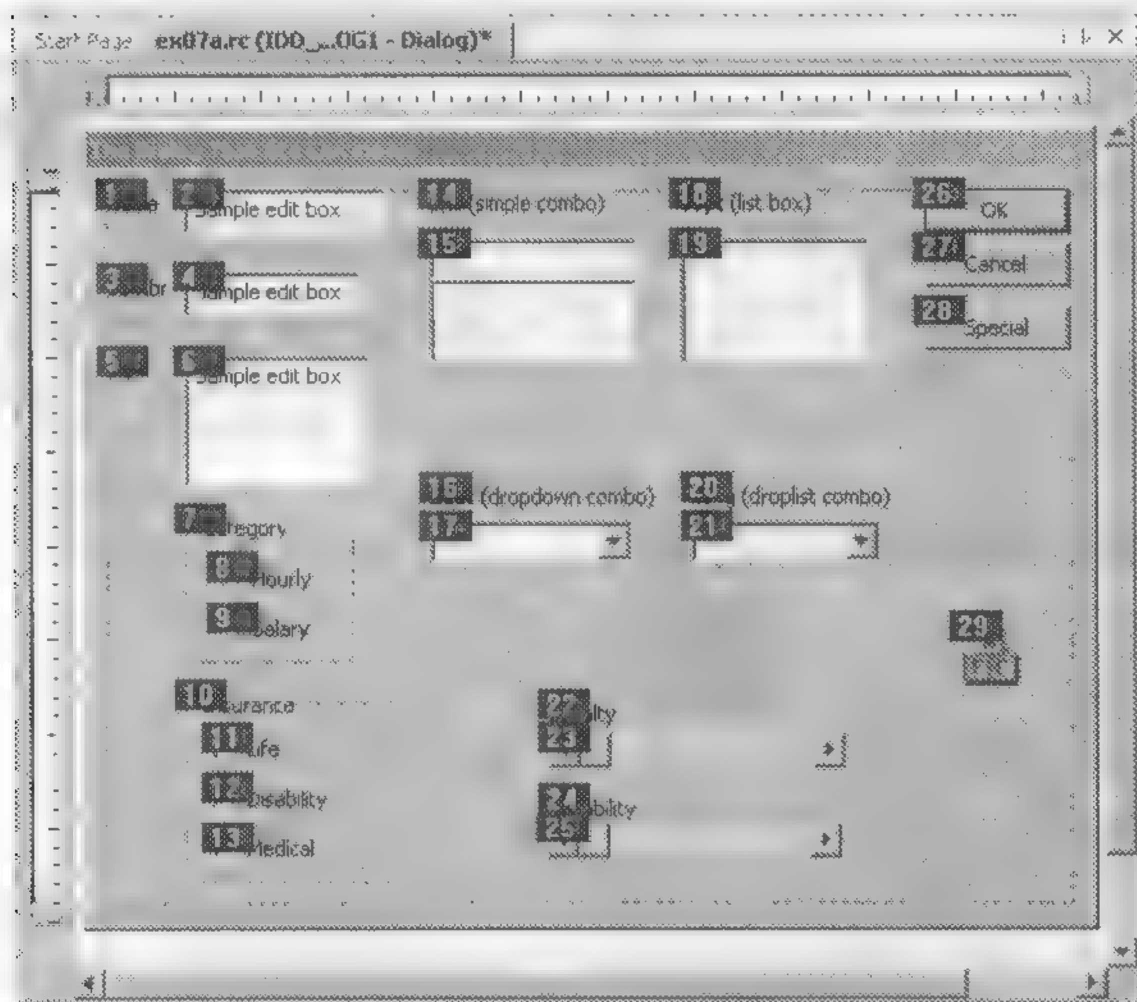
- **Поле со списком Educ (education — образование).** Замените идентификатор на *IDC_EDUC*, установите свойство Sort в FALSE, а у прочих параметров оставьте значения по умолчанию. В свойстве ведите три уровня образования: High School (старшие классы), College (колледж) и Graduate (выпускник), разделив их точкой с запятой (;). В этом поле со списком пользователь сможет набрать все, что ему заблагорассудится, или, щелкнув стрелку, раскрыть список и выбрать нужный элемент (мышью или клавишами-стрелками «вверх» или «вниз»).

Примечание Чтобы задать размер раскрывающейся части поля со списком, щелкните стрелку в правой части поля и мышью сместите нижнюю границу вниз.

- **Список Dept (department — отдел).** Замените идентификатор на *IDC_DEPT*, а у прочих параметров оставьте значения по умолчанию. В этом списке пользователь сможет выбрать лишь один элемент мышью, клавишами-стрелками или введя первый символ нужной строки. Заметьте: у списка нет свойства Data, поэтому вы не сможете ввести начальные значения (как это сделать из программы, вы узнаете чуть позже).
- **Поле со списком Lang (language — язык).** Замените идентификатор на *IDC_LANG* и присвойте свойству Type значение Drop List. Введите в свойство Data названия трех языков: English (английский), French (французский) и Spanish (испанский), разделяя их точкой с запятой (;). В этом поле со списком пользователь сможет выбирать элементы только из раскрывающегося списка. Для этого он должен, щелкнув стрелку, указать нужную строку или ввести ее первую букву и при необходимости уточнить выбор при помощи стрелок.
- **Полосы прокрутки Loyalty (лояльность) и Reliability (надежность).** Не путайте элемент управления «полоса прокрутки» с полосами прокрутки, встроенными в окно. Элемент «полоса прокрутки» ведет себя так же, как и остальные элементы управления, — в частности, в период проектирования его размер можно изменять. Разместите горизонтальные полосы прокрутки, как показано на рис. 7-1 и присвойте им идентификаторы *IDC_LOYAL* и *IDC_RELY*.
- **Кнопки OK, Cancel и Special.** Введите названия кнопок *OK*, *Cancel* и *S&Special*, затем присвойте кнопке Special идентификатор *IDC_SPECIAL*. Чуть позже вы узнаете об особом предназначении идентификаторов по умолчанию *IDOK* и *IDCANCEL*.

- **Произвольный значок (для примера показывается значок MFC).** В диалоговом окне при помощи элемента управления Picture можно отобразить любой значок или растровое изображение, если они определены в описании ресурсов. Воспользуемся MFC-значком программы, обозначенным *IDR_MAINFRAME*. Установите параметр Type в Icon, а Image — в *IDR_MAINFRAME*. Идентификатор оставьте *IDC_STATIC*.

5. **Проверьте порядок перехода между элементами управления.** Выберите из меню Format команду Tab Order. Укажите мышью порядок переключения между элементами управления при нажатии клавиши Tab. Для этого щелкните каждый из элементов управления в следующем порядке и нажмите Enter:



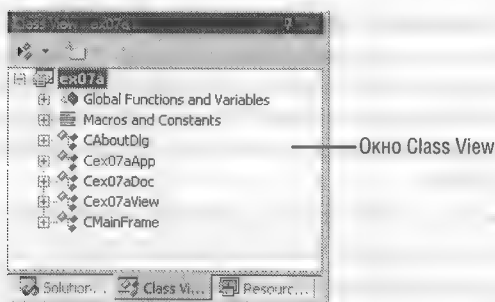
Совет Если вы перепутали порядок переключения, щелкните *последний правильно помеченный* элемент, удерживая клавишу Ctrl.

6. **Сохраните файл ресурсов на диске.** Осторожности ради сохраните файл Ex07a.rc: выберите в меню File команду Save или щелкните одноименную кнопку на панели инструментов. Не закрывайте пока редактор диалоговых окон и оставьте на экране только что созданное диалоговое окно.

Создание класса «диалог»

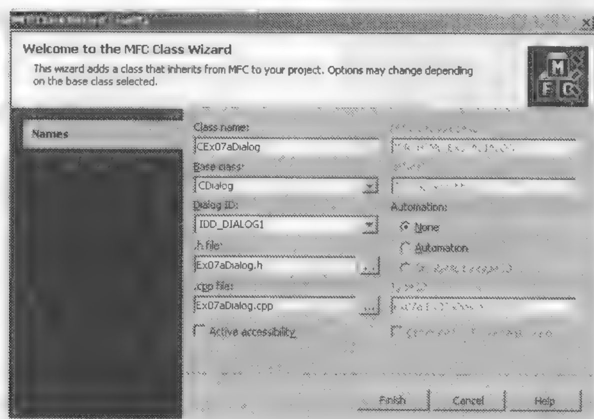
Теперь вы создали диалоговый ресурс, но без соответствующего класса диалогового окна работать с ним нельзя. (О взаимосвязи диалогового окна и стоящих за ним классов см. раздел «Разбор приложения Ex07a».) При создании этого класса используется Class View и редактор диалогов.

1. **Запустите MFC Class Wizard.** В Class View выберите проект Ex07a:



В меню Project выберите команду Add Class (или щелкните правой кнопкой имя проекта в Class View и в контекстном меню последовательно выберите Add и Add Class). В диалоговм окне Add Class выберите шаблон MFC Class и щелкните кнопку OK. Откроется окно мастера MFC Class Wizard.

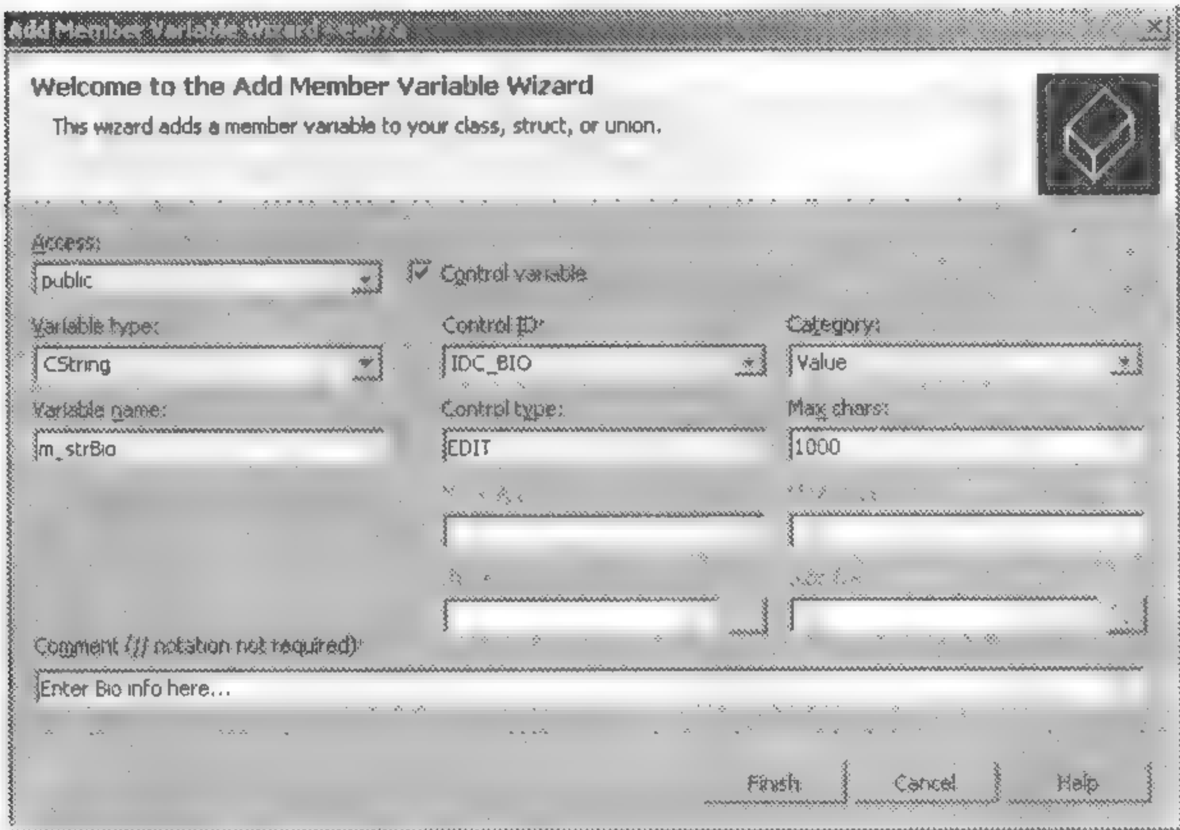
2. **Добавьте класс *CEx07aDialog*.** Создайте класс на основе базового *CDialog*, заполнив поля окна мастера MFC Class Wizard, как показано на рисунке. Не забудьте в поле Dialog ID задать значение *IDD_DIALOG1*, чтобы задействовать уже существующий идентификатор.



Когда вы щелкнете кнопку Finish, в Class View появится класс *CEx07aDialog*, а его CPP-файл откроется в редакторе.

3. **Добавьте переменные класса *CEx07aDialog*.** Добавив класс *CEx07aDialog*, можно перейти к добавлению переменных-членов средствами мастера Add Member Variable Wizard (см. рисунок). Чтобы запустить Add Member Variable Wizard, в Class View щелкните класс *CEx07aDialog* правой кнопкой и в контекстном меню последовательно выберите команды Add и Add Variable.

Вам надо сопоставить переменные-члены каждому элементу управления диалогового окна. Для этого, установив флажок Control Variable, выберите элемент управления в поле со списком Control ID и выберите Value в поле со списком Category. В поле Variable Name ведите имя переменной-члена и определите другие параметры. Вот пример параметров при добавлении переменной-члена m_strBio типа CString для текстового поля Bio:



Повторите процедуру для каждого из элементов управления, перечисленных в следующей таблице. При выборе элементов управления в окне Add Member Variables Wizard можно определить максимальную длину строки или диапазон для числовой переменной. Введите для *IDC_SSN* минимальное значение 0 и максимальное — 999999999.

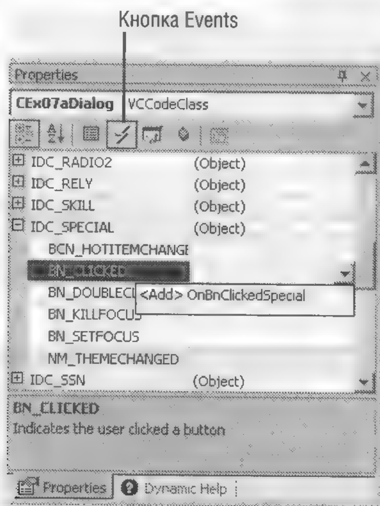
Большинство связей между типами элементов управления и типами переменных очевидно. Однако связь между переключателями и переменными не столь очевидна. С каждой группой переключателей связывают целочисленную переменную, причем первому переключателю соответствует значение 0, второму — 1 и т. д.

Идентификатор элемента управления	Переменная-член	Тип	Параметры
IDC_BIO	m_strBio	CString	Максимальное число символов — 1000
IDC_CAT	m_nCat	int	
IDC_DEPT	m_strDept	CString	
IDC_DIS	m_bInsDis	BOOL	
IDC_EDUC	m_strEduc	CString	
IDC_LANG	m_strLang	CString	
IDC_LIFE	m_bInsLife	BOOL	
IDC_LOYAL	m_nLoyal	int	
IDC_MED	m_bInsMed	BOOL	
IDC_NAME	m_strName	CString	
IDC_RELY	m_nRely	int	
IDC_SKILL	m_strSkill	CString	
IDC_SSN	m_nSsn	int	Минимальное значение — 0, а максимальное — 999999999

4. **Добавьте функцию-обработчик сообщений для кнопки Special.** Классу *CEx07aDialog* не требуется много функций-обработчиков сообщений, так как большую часть работы по управлению диалоговым окном выполняет его базовый класс *CDialog* совместно с Windows. Если вы, например, присвоили иден-

тификатор *IDOK* кнопке OK (по умолчанию), при щелчке этой кнопки вызывается виртуальная функция *OnOK* класса *CDialog*. Однако для других кнопок нужны обработчики сообщений.

При выбранном классе *CEx07aDialog* Class View щелкните кнопку Events в окне Properties, чтобы добавить обработчики. В списке должна быть строка с идентификатором *IDC_SPECIAL*. Разверните узел *IDC_SPECIAL* и щелкните сообщение *BN_CLICKED*. Щелкните стрелку «вниз» рядом с *BN_CLICKED*:

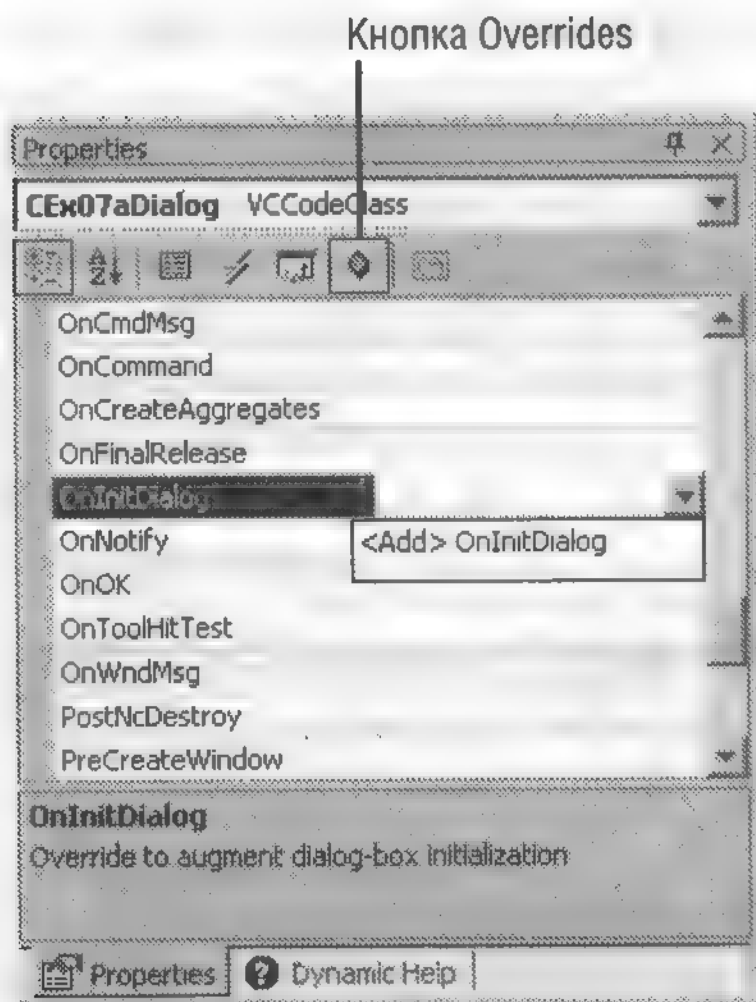


Visual Studio предлагает добавить функцию-обработчик *OnBnClickedSpecial*. Щелкните <Add> OnBnClickedSpecial, чтобы создать ее. Visual Studio откроет в редакторе файл Ex07aDialog.cpp на строке с функцией *OnBnClickedSpecial*. Замените существующий код функции выделенным в листинге оператором *TRACE*:

```
void CEx07aDialog::OnBnClickedSpecial()
{
    TRACE("CEx07aDialog::OnBnClickedSpecial\n");
}
```

5. **Добавьте функцию-обработчик *OnInitDialog*.** Как вы скоро увидите, Visual Studio генерирует код, инициализирующий элементы управления диалогового окна. Однако этот DDX-код (Dialog Data Exchange) не обеспечивает инициализацию элементов списков, поэтому нужно переопределить функцию *CDialog::OnInitDialog*. Хотя *OnInitDialog* — виртуальная функция-член, Visual Studio создаст для нее прототип и заготовку, если вы решите обработать сообщение *WM_INITDIALOG* в производном классе «диалоговое окно».

   Для этого в окне Class View выберите класс *CEx07aDialog*, а в окне Properties щелкните кнопку Overrides. В открывшемся списке выберите функцию *OnInitDialog*, щелкните стрелку «вниз» рядом с ней (см. рисунок).



Выберите команду <Add> OnInitDialog. Visual Studio откроет файл Ex07aDialog.cpp в редакторе и создаст шаблон функции *OnInitDialog*. Замените существующий код выделенным текстом:

```
BOOL CEx07aDialog::OnInitDialog()
{
   // Будьте внимательны: CDialog::OnInitDialog можно вызвать
   // в этой функции только один раз.
   CListBox* pLB = (CListBox*) GetDlgItem(IDC_DEPT);
   pLB->InsertString(-1, "Documentation");
   pLB->InsertString(-1, "Accounting");
   pLB->InsertString(-1, "Human Relations");
   pLB->InsertString(-1, "Security");

   // вызываем после инициализации
   return CDialog::OnInitDialog();
}
```

Этот код инициализирует список Dept с 4 элементами. Для полей со списком вместо инициализации в свойстве Data при желании можно применить такую же процедуру.

Подключение диалогового окна к классу «вид»

Теперь у нас есть ресурс и код для диалогового окна, но окно не подключено к классу «вид». В большинстве приложений диалоговое окно открывается при выборе какой-либо команды из меню, но пока мы меню «не проходили». Поэтому для открытия диалогового окна прибегнем к знакомому сообщению *WM_LBUTTONDOWN*, которое генерируется при щелчке левой кнопки мыши.

1. **Добавьте функцию-член *OnLButtonDown*.** Вы уже проделывали это в предыдущих главах. Просто выделите имя класса *CEx07aView* в Class View и в окне Properties щелкните кнопку Messages. В появившемся списке щелкните стрелку рядом с сообщением *WM_LBUTTONDOWN* и выберите <Add> OnLButtonDown.

2. **Добавьте код функции *OnLButtonDown* в файле Ex07aView.cpp.** Добавьте в заготовку тела функции выделенный код. Большая часть кода состоит из операторов *TRACE*, выводящих значения переменных диалогового окна после того, как пользователь закрыл диалоговое окно. Но главное здесь — вызовы конструктора класса *CEx07aDialog* и функции *DoModal*.

```
void CEx07aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CEx07aDialog dlg;
    dlg.m_strName= "Shakespeare, Will";
    dlg.m_nSsn      = 307806636;
    dlg.m_nCat      = 1; // 0 = почасовая, 1 = оклад
    dlg.m_strBio =  "This person is not a well-motivated tech writer";
    dlg.m_bInsLife  = TRUE;
    dlg.m_bInsDis= FALSE;
    dlg.m_bInsMed= TRUE;
    dlg.m_strDept= "Documentation";
    dlg.m_strSkill  = "Writer";
    dlg.m_nLang     = 0;
    dlg.m_strEduc= "College";
    dlg.m_nLoyal = dlg.m_nRely= 50;
    int ret         = dlg.DoModal();
    TRACE("DoModal return = %d\n", ret);
    TRACE("name = %s, ssn = %d, hourly = %d salary = %d\n",
        dlg.m_strName, dlg.m_nSsn, dlg.m_nCat);
    TRACE("dept = %s, skill = %s, lang = %d, educ = %s\n",
        dlg.m_strDept, dlg.m_strSkill, dlg.m_nLang, dlg.m_strEduc);
    TRACE("life = %d, dis = %d, med = %d, bio = %s\n",
        dlg.m_bInsLife, dlg.m_bInsDis, dlg.m_bInsMed, dlg.m_strBio);
    TRACE("loyalty = %d, reliability = %d\n",
        dlg.m_nLoyal, dlg.m_nRely);
}
```

3. **Добавьте код в виртуальную функцию *OnDraw* в файл Ex07aView.cpp.** Чтобы предложить пользователю нажать левую кнопку мыши, добавьте в класс *CEx07aView* функцию *OnDraw* (ее заготовку сгенерировал мастер MFC Application Wizard). Замените существующий код выделенным:

```
void CEx07aView::OnDraw(CDC* pDC)
{
    CEx07aDoc* pDoc = GetDocument();
    ASSERT_VALID (pDoc);
    pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

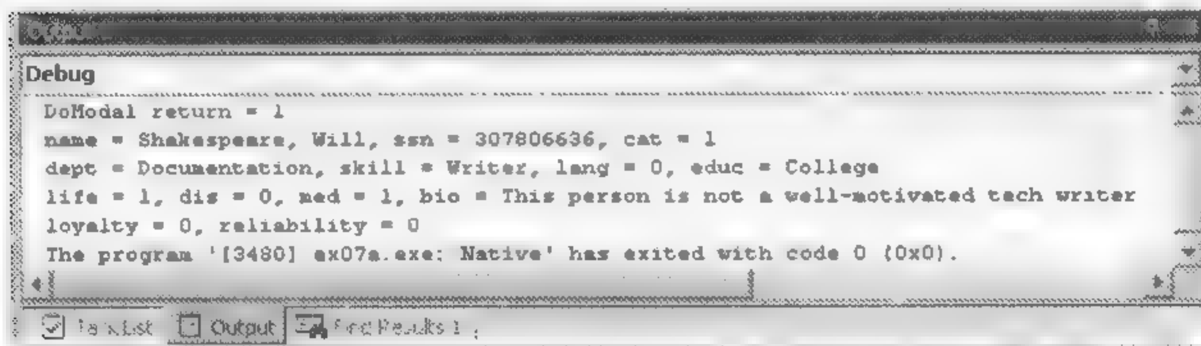
4. **Добавьте в файл Ex07aView.cpp оператор включения класса «диалоговое окно».** Показанная выше функция *OnLButtonDown* зависит от объявления класса *CEx07aDialog*. Вы должны включить оператор:

```
#include "Ex07aDialog.h"
```

в начало файла с исходным кодом для класса *CEx07aView* (Ex07aView.cpp) после оператора:

```
#include "Ex07aView.h"
```

5. **Соберите и протестируйте приложение.** Если все сделано правильно, вы сможете собрать и запустить приложение Ex07a из Visual C++. Попробуйте, вводя данные в каждый элемент управления, щелкать кнопку OK и наблюдать за выводом операторов *TRACE* в окне Output. Полосы прокрутки пока ничего особенного не делают — о них речь пойдет позже. Обратите внимание и на то, что происходит при нажатии клавиши Enter в момент ввода текстовых данных в каком-либо элементе управления — диалоговое окно сразу закрывается. Вот пример трассировочной информации в окне Output:



Разбор приложения Ex07a

После вызова *DoModal* управление возвратится в программу только после закрытия диалогового окна. Если вам это ясно, значит, вы поняли, что такое модальное диалоговое окно. Перейдя к работе с немодальными диалоговыми окнами, вы еще оцените простоту программирования модальных диалоговых окон, потому что при вызове *DoModal* очень многое остается за кадром. Но вернемся к нашей теме и рассмотрим краткую сводку «кто кого вызывает»:

```
CDialog::DoModal
    CEx07aDialog::OnInitDialog
        ...дополнительная инициализация...
        CDialog:: OnInitDialog
            CWnd::UpdateData(FALSE)
                CEx07aDialog::DoDataExchange
    пользователь вводит данные...
    пользователь щелкает кнопку OK
    CEx07aDialog::OnOK
        ...дополнительная проверка...
        CDialog::OnOK
            CWnd::UpdateData(TRUE)
                CEx07aDialog::DoDataExchange
            CDialog::EndDialog(IDOK)
```

OnInitDialog и *DoDataExchange* — виртуальные функции, переопределенные в классе *CEx07aDialog*. Windows вызывает *OnInitDialog* при инициализации диалогового окна, что приводит к вызову *DoDataExchange* — виртуальной функции класса *CWnd*, переопределенной в Visual Studio. Взгляните на листинг этой функции:

```
void CEx07aDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_BIO, m_strBio);
    DDV_MaxChars(pDX, m_strBio, 1000);
```

```
    DDX_Radio(pDX, IDC_CAT, m_nCat);
    DDX_LBString(pDX, IDC_DEPT, m_strDept);
    DDX_Check(pDX, IDC_DIS, m_bInsDis);
    DDX_CBString(pDX, IDC_EDUC, m_strEduc);
    DDX_CBIndex(pDX, IDC_LANG, m_nLang);
    DDX_Check(pDX, IDC_LIFE, m_bInsLife);
    DDX_Scroll(pDX, IDC_LOYAL, m_nLoyal);
    DDX_Check(pDX, IDC_MED, m_bInsMed);
    DDX_Text(pDX, IDC_NAME, m_strName);
    DDX_Scroll(pDX, IDC_RELY, m_nRely);
    DDX_CBString(pDX, IDC_SKILL, m_strSkill);
    DDX_Text(pDX, IDC_SSN, m_nSsn);
    DDV_MinMaxInt(pDX, m_nSsn, 0, 999999999);
}
```

DoDataExchange и функции *DDX_* (*обмен*) и *DDV_* (*проверка корректности данных*) — «двусторонние». Если *UpdateData* вызывается с параметром *FALSE*, то она переносит данные из переменных-членов в элементы управления диалогового окна, а если с параметром *TRUE*, то направляет данные из элементов управления в переменные-члены. *DDX_Text* переопределяется для адаптации ко всему имеющемуся множеству типов данных.

Функция *EndDialog* играет главную роль в процедуре завершения диалогового окна. *DoModal* возвращает параметр, передаваемый в *EndDialog*. *IDOK* принимает данные из диалогового окна, а *IDCANCEL* отменяет диалоговое окно.

Совет Вы можете написать свою «пользовательскую» DDX-функцию и подключить к Visual C++. Это бывает удобно, если во всей программе вы применяете уникальный тип данных. Подробнее см. техническую рекомендацию «TN026: DDX and DDV Routines» в интерактивной справочной системе.

Усовершенствование программы Ex07a

Программа Ex07a, обладая массой возможностей, не требовала от программиста особых усилий. Теперь создадим новую версию с расширенной функциональностью. Избавим Ex07a от скверной привычки завершать свою работу при нажатии клавиши Enter и введем поддержку полос прокрутки.

Перехват управления при выходе по *OnOK*

В исходной программе Ex07a виртуальная функция *CDialog::OnOK* обрабатывала щелчок кнопки OK, что запускало обмен данными и процедуру завершения диалогового окна. Как оказалось, клавиша Enter давала тот же результат — может быть, это как раз то, что нужно, однако иногда такое поведение неприемлемо. Если пользователь нажмет Enter, скажем, при вводе в поле Name, его сразу «выбросит» из диалогового окна. Вряд ли это ему понравится.

Что же происходит? В момент нажатия этой клавиши Windows ищет кнопку, на которой установлен *фокус ввода* (input focus) — на экране он выглядит как

пунктирная рамка. Если фокус не установлен ни на одну кнопку, Windows ищет указанную программой или в ресурсе *кнопку по умолчанию* (default pushbutton) — у такой кнопки более толстый контур. Если и этой кнопки не обнаруживается, вызывается виртуальная функция *OnOK* — даже когда в окне нет кнопки OK.

Клавишу Enter можно отключить, просто написав «пустую» функцию *CEx07aDialog::OnOK* и добавив код завершения в новую функцию, реагирующую на щелчок кнопки OK. Последовательность действий такова.

1. **Сопоставьте кнопку IDOK виртуальной функции *OnOK*.** В окне Class View выберите класс *CEx07aDialog*, а в окне Properties — кнопку Overrides. В открывшемся списке выберите *OnOK*, щелкните стрелку «вниз» рядом с ней и выберите <Add> OnOK. Visual Studio создаст прототип и заготовку для функции *OnOK*.
2. **В редакторе диалогов, измените идентификатор кнопки OK.** Выберите кнопку OK, измените ее идентификатор с *IDOK* на *IDC_OK* и установите в FALSE свойство Default Button.
3. **Создайте функцию-член *OnClickedOk*.** В окне Class View выберите класс *CEx07aDialog*, а в окне Properties — кнопку Events. Раскройте узел *IDC_OK*, выберите сообщение *BN_CLICKED* и, щелкнув стрелку «вниз» рядом с ней, выберите <Add> OnBnClickedOk.
4. **Отредактируйте тело функции *OnClickedOk* в Ex07aDialog.cpp.** Она вызывает функцию *OnOK* базового класса, как это делала исходная функция *CEx07aDialog::OnOK*. Вот ее код:

```
void CEx07aDialog::OnClickedOk()
{
    TRACE("CEx07aDialog::OnClickedOk\n");
    CDialog::OnOK();
}
```

5. **Отредактируйте исходную функцию *OnOK* в Ex07aDialog.cpp.** Эта функция — «пустой» обработчик кнопки с прежним идентификатором *IDOK*. Модифицируйте код:

```
void CEx07aDialog::OnOK()
{
    // заглушка для функции OnOK - НЕ вызывайте CDialog::OnOK()
    TRACE("CEx07aDialog::OnOK\n");
}
```

6. **Соберите и протестируйте приложение.** Попробуйте теперь нажать клавишу Enter. Ничего произойти не должно, но в окне Debug появится вывод оператора *TRACE*. Однако щелчок кнопки OK должен, как и раньше, завкрывать диалоговое окно.

Обработка *OnCancel*

Так же, как нажатие клавиши Enter приводило к вызову *OnOK*, нажатие клавиши Esc инициирует вызов *OnCancel*, в результате чего диалоговое окно завершается с кодом возврата *IDCANCEL* из функции *DoModal*. Программа Ex07a не предусматривает особой обработки *IDCANCEL*, поэтому нажатие Esc (или щелчок кнопки

Close) закрывает диалоговое окно. Вы можете обойти этот процесс, применив функцию-заглушку *OnCancel* по аналогии с тем, что уже делалось для кнопки OK.

Подключение полос прокрутки

Графический редактор позволяет размещать в диалоговом окне элементы управления «полосы прокрутки», а Add Member Variable Wizard — создавать для них целочисленные переменные-члены. Чтобы полосы прокрутки Loyalty и Reliability заработали, надо дополнить программу соответствующим кодом.

Эти элементы управления позволяют считывать и записывать текущую позицию бегунка и границы заданного диапазона. Если установить диапазон, скажем, от 0 до 100, соответствующая переменная-член со значением 50 поместит движок в центр полосы прокрутки. (Функция *CScrollBar::SetScrollPos* тоже задает позицию движка на полосе прокрутки.) Когда пользователь перемещает движок или щелкает стрелки, полоса прокрутки отправляет в диалоговое окно сообщения *WM_HSCROLL* и *WM_VSCROLL*. Обработчики сообщений в диалоговом окне расшифровывают эти сообщения и изменяют позицию движка на полосе прокрутки.

Особенность этих элементов в том, что все горизонтальные полосы посылают одно сообщение — WM_HSCROLL, а все вертикальные — WM_VSCROLL. А раз в нашем «навороченном» диалоговом окне две горизонтальные полосы прокрутки, значит, один-единственный обработчик сообщения WM_HSCROLL должен как-то различать, от какой пришло сообщение.

Добавим в программу Ex07a логику управления полосами прокрутки.

1. **Добавьте операторы *enum*, чтобы задать предельные значения диапазона прокрутки.** Включите в самое начало объявления класса в файле Ex07aDialog.h строки:

```
enum { nMin = 0 };
enum { nMax = 100 };
```

2. **Отредактируйте функцию *OnInitDialog*, чтобы инициализировать границы диапазона прокрутки.** Эта функция должна устанавливать минимальное и максимальное значения диапазона прокрутки так, чтобы переменные-члены *CEx07aDialog* отражали величины, выраженные в процентах: 100 означает «установить движок в крайнюю правую позицию», а 0 — «установить движок в крайнюю левую позицию».

   Добавьте в файле Ex07aDialog.cpp в функцию-член *OnInitDialog* класса *CEx07aDialog* такой код:

```
CScrollBar* pSB = (CScrollBar*) GetDlgItem(IDC_LOYAL);
pSB->SetScrollRange(nMin, nMax);
pSB = (CScrollBar*) GetDlgItem(IDC_RELY);
pSB->SetScrollRange(nMin, nMax);
```

3. **Добавьте в *CEx07aDialog* обработчик сообщений от полос прокрутки.** В окне Class View выберите класс *CEx07aDialog*, а затем в окне Properties щелкните кнопку Messages. Выберите сообщение *WM_HSCROLL* и добавьте функцию-член *OnHScroll*. Введите выделенный код:

```
void CEx07aDialog::OnHScroll(UINT nSBCode, UINT nPos,
                                              CScrollBar* pScrollBar)
{
    int nTemp1, nTemp2;

    nTemp1 = pScrollBar->GetScrollPos();
    switch(nSBCode) {
    case SB_THUMBPOSITION:
        pScrollBar->SetScrollPos(nPos);
        break;
    case SB_LINELEFT: // кнопка "стрелка-влево"
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 - nTemp2) > nMin) {
            nTemp1 -= nTemp2;
        }
        else {
            nTemp1 = nMin;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    case SB_LINERIGHT: // кнопка "стрелка-вправо"
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 + nTemp2) < nMax) {
            nTemp1 += nTemp2;
        }
        else {
            nTemp1 = nMax;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    }
}
```

4. **Соберите и протестируйте приложение.** Вновь соберите и запустите программу Ex07a. Заработали ли полосы прокрутки? Бегунки должны перемещаться, когда вы щелкаете стрелки на полосах прокрутки, а также подчиняться перетаскиванию. (Заметьте: пока в программу не включена логика, позволяющая реагировать на щелчок пользователем самой полосы прокрутки.)

Доступ к элементам управления: *CWnd*-указатели и идентификаторы

Размечая диалоговый ресурс в графическом редакторе, вы определяете элементы управления при помощи идентификаторов, таких как *IDC_SSN*. Однако в программном коде бывает нужен доступ к стоящему за элементом управления оконному объекту. Поэтому в MFC-библиотеке предусмотрена функция *CWnd::GetDlgItem*, преобразующая идентификатор в *CWnd*-указатель. Вы уже видели такое преобразование в функциях-членах *OnInitDialog* и *OnClickedOk* класса *CEx07aDialog*. Каркас приложений чудесным образом «создавал» этот *CWnd*-указатель, потому что

там не было предусмотрено вызова конструктора для объектов управления. Этот указатель временный, и сохранять его нельзя.

Совет Чтобы преобразовать *CWnd*-указатель в идентификатор элемента управления, задействуйте функцию-член *GetDlgCtrlID* класса *CWnd*.

Фон диалогового окна и цвет элементов управления

Чтобы изменить фон или отдельные элементы управления в диалоговом окне, придется потрудиться. Каждый элемент управления, прежде чем появиться на экране, посылает родительскому диалоговому окну сообщение *WM_CTLCOLOR*. Такое же сообщение отправляется и самому диалоговому окну. Если написать в производном классе диалогового окна обработчик этого сообщения, можно установить цвет текста и его фон, а также выбрать кисть для заполнения нетекстовой области элемента управления или диалогового окна.

Вот пример функции *OnCtlColor*, задающей желтый фон для всех полей ввода и красный фон для диалогового окна. Переменные *m_hYellowBrush* и *m_hRedBrush* — это переменные-члены типа *HBRUSH*, инициализируемые в *OnInitDialog*. Параметр *nCtrColor* определяет тип элемента управления, а *pWnd* — конкретный элемент управления. Чтобы установить цвет одного отдельно взятого поля ввода, нужно преобразовать *pWnd* в идентификатор дочернего окна и протестировать его.

```
HBRUSH CMyDialog::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
   if (nCtlColor == CTLCOLOR_EDIT) {
      pDC->SetBkColor(RGB(255, 255, 0)); // желтый
      return m_hYellowBrush;
   }
   if (nCtlColor == CTLCOLOR_DLG) {
      pDC->SetBkColor(RGB(255, 0, 0)); // красный
      return m_hRedBrush;
   }
   return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
}
```

Примечание Диалоговое окно не размещает WM_CTLCOLOR в очереди сообщений; вместо этого оно — чтобы сразу отправить сообщение — вызывает Win32-функцию *SendMessage*. Это позволяет обработчику сообщения вернуть параметр, в данном случае описатель кисти. Это не MFC-объект *CBrush*, а Win32-объект *HBRUSH*. Создать кисть можно, вызвав Win32-функции *CreateSolidBrush*, *CreateHatchBrush* и др.

Добавление элементов управления во время исполнения

Мы уже видели, как на этапе проектирования с помощью редактора диалоговых окон создаются элементы управления диалогового окна. Если вам надо добавить элемент управления в период выполнения, придерживайтесь следующей схемы.

1. **Добавьте в свой класс «диалоговое окно» переменную-член для внедряемого элемента управления.** Существует несколько MFC-классов таких элементов управления: *CButton*, *CEdit*, *CListBox* и *CComboBox*. Внедряемые объекты C++ конструируются и уничтожаются вместе с объектом «диалоговое окно».
2. **Добавьте константу-идентификатор нового элемента управления.** Щелкните правой кнопкой ресурс диалога в Resource View и в контекстном меню выберите Resource Symbols — откроется одноименное диалоговое окно. Добавьте новую константу.
3. **С помощью окна Properties утилиты Class View переопределите функцию *CDialog::OnInitDialog*.** Эта функция должна вызывать функцию-член *Create* внедряемого элемента управления. В результате такого вызова в диалоговом окне появится новый элемент. Windows уничтожит окно этого элемента управления при уничтожении диалогового окна.
4. **Добавьте вручную в производный класс диалогового окна обработчики уведомляющих сообщений для нового элемента управления.**

В главе 12 мы попробуем в период выполнения добавить в диалоговое окно *поле ввода с форматированием* (rich edit control).

Другие возможности элементов управления

Вы уже видели, как путем добавления кода в функцию-член *OnInitDialog* класса диалогового окна настраивается класс элементов управления *CScrollBar*. Другие элементы управления программируются аналогично. Посмотрите в справочнике *Microsoft Foundation Class Library Reference* список классов, относящихся к элементам управления, в частности, обратите внимание на классы *CListBox* и *CComboBox*. Каждый обладает рядом свойств, которые окно Properties и мастера Visual Studio напрямую не поддерживают. В частности, некоторые поля со списком поддерживают выделение сразу нескольких строк при применении стиля *LBS_MULTIPLESEL*. Если вы хотите задействовать эти свойства, не пытайтесь добавлять соответствующие переменные-члены через Class View, а определите сами нужные переменные-члены и дополните функции *OnInitDialog* и *OnClickedOk* своим кодом, отвечающим за обмен нужной информацией.

Стандартные диалоговые окна Windows

Windows предоставляет набор стандартных диалоговых окон; их поддерживают и MFC-классы (они находятся в файле comdlg32.dll). Вы, вероятно, знакомы с этими окнами (всеми или некоторыми), поскольку они используются во многих Windows-приложениях, включая Visual C++. Все классы стандартных диалоговых окон произведены от одного базового класса *CCommonDialog* (табл. 7-1).

Таблица 7-1. Классы семейства *CCommonDialog*

Класс	Назначение
CColorDialog	Выбор или создание цвета
CFileDialog	Открытие или сохранение файла
CFindReplaceDialog	Замена в документе одного текста другим
CFontDialog	Выбор шрифта из списка доступных шрифтов
COleDialog	Применяется для внедрения объектов OLE
CPageSetupDialog	Ввод параметров страницы документа
CPrintDialog	Настройка принтера и печать документа
CPrintDialogEx	Печать и предварительный просмотр перед печатью в Windows 2000

У всех стандартных диалоговых окон общая особенность: они принимают информацию у пользователя, но ничего с ней не делают. Скажем, диалоговое окно File Open помогает пользователю открыть файл, но на деле лишь сообщает программе путь и имя выбранного файла — об остальном должна позаботиться сама программа. Примерно так же действует и диалоговое окно, предназначенное для выбора шрифта: в него вводят параметры шрифта, но оно не создает шрифта.

Прямое использование класса *CFileDialog*

Открыть файл с помощью этого класса очень просто. Вот пример кода, открывающего файл, выбранный пользователем в диалоговом окне:

```
CFileDialog dlg(TRUE, "bmp", "*.bmp");
if (dlg.DoModal() == IDOK) {
    CFile file;
    VERIFY(file.Open(dlg.GetPathName(), CFile::modeRead));
}
```

Первый параметр (TRUE) конструктора указывает, что данный объект — диалоговое окно открытия (File Open), а не сохранения (File Save) файла; «bmp» — это расширение файлов по умолчанию, и **.bmp* появится в окне ввода имени файла. Функция *CFileDialog::GetPathName* возвращает объект *CString*, который содержит полное имя выбранного файла (с указанием пути).

Производные классы стандартных диалоговых окон

Чаще всего классы стандартных диалоговых окон вы будете использовать напрямую. Если же вы решите создать свои производные классы, то сможете расширить функциональность стандартных диалоговых окон, не дублируя их код. Однако у каждого диалогового окна COMDLG32 своя специфика. И хотя в следующем примере мы будем иметь дело с диалоговым окном, предназначенным для работы с файлами, он все же даст вам представление об адаптации других стандартных диалоговых окон.

Вложение диалоговых окон

Win32 позволяет «вкладывать» диалоговые окна друг в друга и тем самым выводить на экран несколько диалоговых окон как единое целое. Сначала нужно со-

здать шаблон диалогового ресурса с «дырой» в нем — обычно это элемент управления «группирующая рамка» — и присвоить дочернему окну характерный идентификатор *stc32* (=0x045f). Далее программа должна установить ряд параметров, которые подскажут COMDLG32, что нужно задействовать именно этот шаблон. Кроме того, программа должна поставить *ловушку* (hook) в цикле выборки сообщений в COMDLG32, чтобы получить «приоритетный» доступ к определенным сообщениям. Проделав эти операции, вы получите диалоговое окно, все еще дочернее по отношению к диалоговому окну COMDLG32, несмотря даже на то, что ваш шаблон — это «обертка» шаблона COMDLG32. Все это выглядит сложно и на самом деле окажется сложным, если вы не используете MFC. Работая с MFC, вы создаете шаблон диалогового ресурса и, как уже было сказано, формируете класс, производный от одного из базовых классов стандартных диалоговых окон, включаете в *OnInitDialog* связующий код, специфичный для конкретного класса, а затем просто в окне Properties создаете обработчики сообщений, поступающих от новых элементов управления вашего шаблона. И все.

Программа-пример Ex07b: использование класса *CFileDialog*

Вы создадите класс, наследующий классу *CEx07bDialog*, который добавит в стандартное диалоговое окно для работы с файлами кнопку Delete all matching files («удалить все аналогичные файлы»). Кроме того, он изменит заголовок диалогового окна и заменит кнопку Open на Delete (для удаления отдельных файлов). Вы научитесь использовать вложенные диалоговые окна, чтобы добавлять в стандартные диалоговые окна новые элементы управления. Обновленное диалоговое окно активизируется так же, как и в Ex07a, — простым щелчком в границах окна представления. Поскольку вы уже должны уметь работать с Visual C++, мы не будем так подробно, как раньше, описывать подготовительные операции. Вот как выглядит диалоговое окно, создаваемое программой (рис. 7-2):

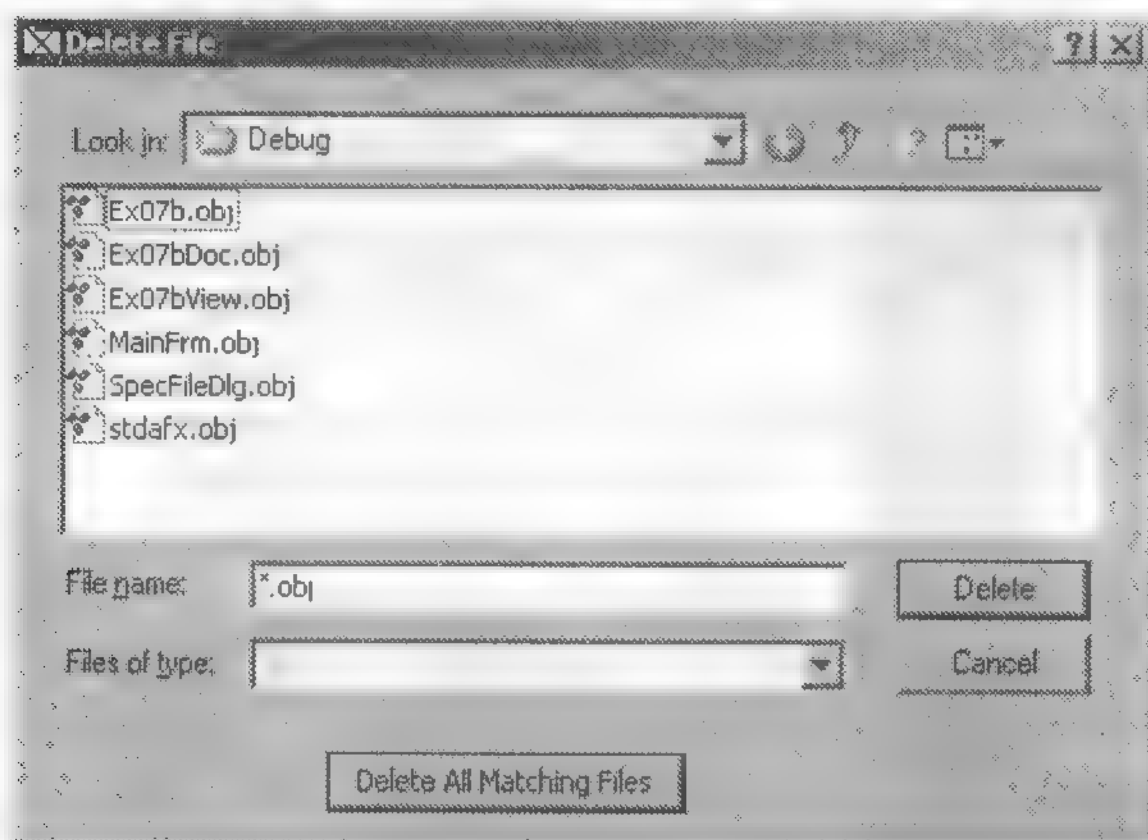


Рис. 7-2. *Диалоговое окно Delete File в действии*

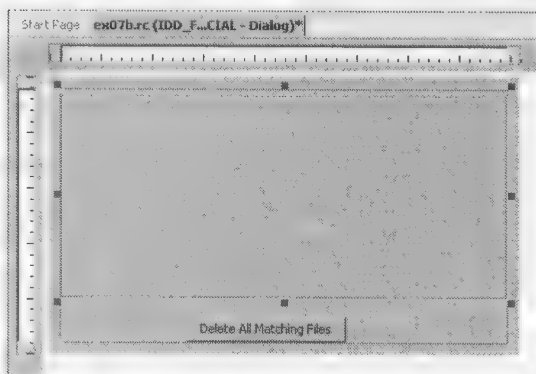
Итак, создаем приложение Ex07b.

1. **Запустите MFC Application Wizard и создайте проект Ex07b.** На странице Application Type мастера установите переключатель в положение Single

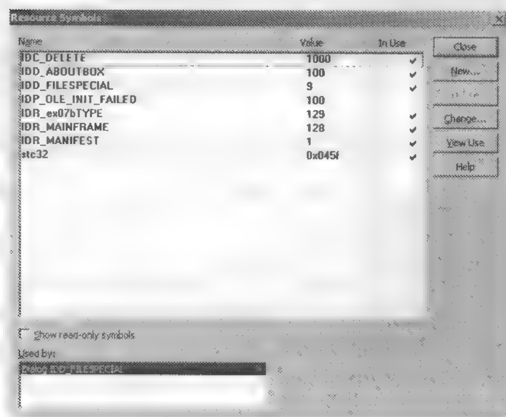
document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.

2. **Создайте новый диалоговый ресурс.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс. Установите размер диалогового окна равным 3×5 дюймов и присвойте ему идентификатор *IDD_FILESPECIAL*. Свойству Style присвойте значение Child, свойству Border — None и установите свойства Clip Siblings и Visible в TRUE.

3. **Создайте элементы управления диалогового окна.** Удалите кнопки OK и Cancel. Создайте кнопку в нижней части диалогового окна с идентификатором *IDC_DELETE*. Установите свойство Caption в Delete All Matching Files. Создайте группирующую рамку, в которой присвойте идентификатору значение *stc32=0x045f*, а свойству Visible — False:

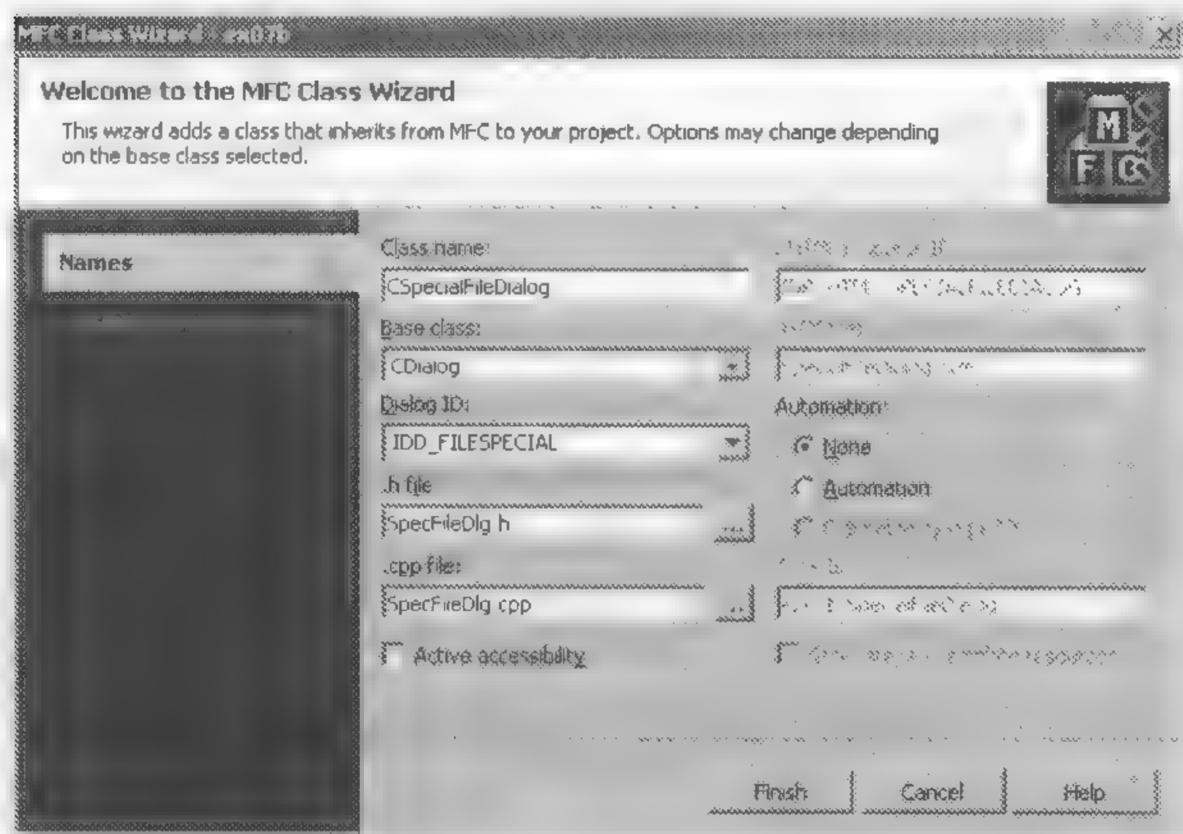


Проверьте результат своих трудов, щелкнув правой кнопкой диалоговый ресурс *IDD_FILESPECIAL* в окне Resource View и выбрав в контекстном меню Resource Symbols. Список символов должен выглядеть так:



4. **Используя MFC Class Wizard, создайте класс *CSpecialFileDialog*.** В окне Class View щелкните правой кнопкой проект Ex07b и в контекстном меню последовательно выберите Add и Add Class. В открывшемся диалоговом окне выбери-

те шаблон MFC Class и щелкните кнопку Open, чтобы запустить MFC Class Wizard. Заполните поля мастера, как показано на рисунке. Не забудьте изменить имена файлов на SpecFileDlg.h и SpecFileDlg.cpp. К сожалению, мы не сможем выбрать *CFileDialog* в качестве базового класса в списке Base Class — это отсоединит класс от шаблона *IDD_FILESPECIAL*. Вместо этого придется выбрать *CDialog* и сделать замену вручную. Закончив настройку, щелкните кнопку Finish.



5. **Отредактируйте файл SpecFileDlg.h.** Замените строку:

```
class CSpecialFileDialog : public CDialog
```

на:

```
class CSpecialFileDialog : public CFileDialog
```

Кроме того, добавьте две открытых переменных-члена класса:

```
CString m_strFilename;
BOOL m_bDeleteAll;
```

И, наконец, отредактируйте объявление конструктора:

```
CSpecialFileDialog(BOOL bOpenFileDialog,
    LPCTSTR lpszDefExt = NULL,
    LPCTSTR lpszFileName = NULL,
    DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    LPCTSTR lpszFilter = NULL,
    CWnd* pParentWnd = NULL);
```

6. **Замените *CDialog* на *CFileDialog* в файле SpecFileDlg.cpp.** Для этого в меню Edit последовательно выберите Find And Replace и Replace и замените это имя глобально.
7. **Отредактируйте конструктор *CSpecialFileDialog* в файле SpecFileDlg.cpp.** Конструктор производного класса должен вызывать конструктор базового класса и инициализировать переменную-член *m_bDeleteAll*. Кроме того, он должен устанавливать некоторые поля в переменной-члене *m_ofn* (базового класса *CFileDialog*), которая является экземпляром Win32-структуры *OPENFILENAME*. Поля *Flags* и *lpTemplateName* структуры управляют подключением

класса к шаблону *IDD_FILESPECIAL*, а поле *lpstrTitle* изменяет заголовок основного диалогового окна. Отредактируйте конструктор:

```
CSpecialFileDialog::CSpecialFileDialog(BOOL  bOpenFileDialog,
      LPCTSTR lpszDefExt, LPCTSTR lpszFileName, DWORD dwFlags,
      LPCTSTR lpszFilter, CWnd* pParentWnd)
   : CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName,
      dwFlags, lpszFilter, pParentWnd)
{
   m_ofn.Flags |= OFN_ENABLETEMPLATE;
   m_ofn.lpTemplateName = MAKEINTRESOURCE(IDD_FILESPECIAL);
   m_ofn.lpstrTitle = "Delete File";
   m_bDeleteAll = FALSE;
}
```

8. **Переопределите функцию OnInitDialog в классе *CSpecialDialog*.** Для этого в окне Class View выберите класс *CSpecialFileDialog*, а в окне Properties щелкните кнопку Overrides и добавьте функцию *OnInitDialog*. Функция-член *OnInitDialog* должна заменить кнопку Open в стандартном диалоговом окне на кнопку Delete. Идентификатор кнопки — *IDOK*. Отредактируйте код так:

```
BOOL CSpecialFileDialog::OnInitDialog()
   BOOL bRet = CFileDialog::OnInitDialog();
   if (bRet == TRUE) {
      GetParent()->GetDlgItem(IDOK)->SetWindowText("Delete");
   }
return bRet;
}
```

9. **Создайте обработчик новой кнопки *IDC_DELETE* (Delete All Matching Files) в классе *CSpecialFileDialog*.** Для этого в окне Class View выберите класс *CSpecialFileDialog*, в окне Properties щелкните кнопку Events, разверните узел IDC_DELETE и добавьте функцию *OnBnClickedDelete*. Функция-член *OnBnClickedDelete* устанавливает флажок *m_bDeleteAll*, а потом заставляет основное диалоговое окно вернуть управление, как будто была нажата кнопка Cancel. Клиентская программа (в данном случае объект «вид») получает *IDCANCEL*, возвращаемый из *DoModal*, и проверяет флажок, чтобы узнать, надо ли удалить все файлы:

```
void CSpecialFileDialog::OnBnClickedDelete()
{
   m_bDeleteAll = TRUE;
   // 0x480 - идентификатор дочернего окна поля ввода File Name[1]
   // (по данным, полученным от SPYXX)
   GetParent()->SetDlgItem(0x480)->GetWindowText(m_strFileName);
   GetParent()->SendMessage(WM_COMMAND, IDCANCEL);
}
```

[1] Идентификаторы дочерних элементов управления стандартных диалоговых окон перечислены в файле dlgs.h в каталоге Include. В частности, для поля ввода File Name используется #define-константа edt1. В этом же файле объявлена #define-константа stc32. — *Прим. перев.*

10. **Добавьте код виртуальной функции *OnDraw* в файл Ex07bView.cpp.** Чтобы функция *OnDraw* класса *CEx07bView* (заготовку которой генерирует мастер MFC Application Wizard) предлагала пользователю нажать кнопку мыши, закодируем ее так:

```
void CEx07bView::OnDraw(CDC* pDC)
{
    CEx07bDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

11. **Добавьте в класс *CEx07bView* обработчик сообщения *WM_LBUTTONDOWN*.** Выделите имя класса *CEx07bView* в Class View и в окне Properties щелкните кнопку Messages. В появившемся списке найдите сообщение *WM_LBUTTONDOWN* и, щелкнув стрелку рядом с ним, выберите <Add> OnLButtonDown.

```
void CEx07bView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CSpecialFileDialog dlgFile(TRUE, NULL, "*.obj");
    CString strMessage;
    int nModal = dlgFile.DoModal();
    if ((nModal == IDCANCEL) && (dlgFile.m_bDeleteAll)) {
        strMessage.Format(
                "Are you sure you want to delete all %s files?",
                dlgFile.m_strFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES) {
            HANDLE h;
            WIN32_FIND_DATA fData;
            while ((h = ::FindFirstFile(dlgFile.m_strFilename, &fData))
                    != (HANDLE) 0xFFFFFFFF) { // MFC-эквивалента нет
                if (::DeleteFile(fData.cFileName) == FALSE) {
                    strMessage.Format(Unable to delete file %s\n",
                        fData.cFileName);
                    AfxMessageBox(strMessage);
                    break;
                }
            }
        }
    }
    else if (nModal == IDOK) {
        CString strSingleFilename = dlgFile.GetPathName();
        strMessage.Format(
            "Are you sure you want to delete %s?", strSingleFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES) {
            CFile::Remove(strSingleFilename);
        }
    }
}
```

Как вы помните, стандартные диалоговые окна всего лишь собирают данные и не более того. Поскольку объект «вид» является клиентом объекта «диа-

логовое окно», он должен вызывать *DoModal* для активизации *объекта «выбор файлов»* (file dialog object), а затем разбираться, что делать с полученными от него данными. В нашем случае у него есть значение, возвращенное функцией *DoModal* (*IDOK* или *IDCANCEL*), значение открытой переменной-члена — флажка *m_bDeleteAll* и информация от ряда функций-членов класса *CFileDialog* (скажем, *GetPathName*). Если *DoModal* возвращает *IDCANCEL*, а флажок — *TRUE*, он обращается к файловой системе Win32 и делает вызовы, необходимые для удаления всех файлов с выбранным расширением. Если же *DoModal* возвращает *IDOK*, то для удаления отдельного файла функция может задействовать функции класса *CFile* из библиотеки MFC.

Применение глобальной функции *AfxMessageBox* — удобный способ вызова простого диалогового окна, в котором отображается какой-то текст и который запрашивает у пользователя ответ типа «да-нет». Все варианты подобных диалоговых окон — их обычно называют *информационными* (message boxes) — и их параметры описаны в документации к Visual Studio.

12. **Включите заголовочный файл *SpecFileDlg.h* в Ex07bView.cpp.** Вам понадобится оператор:

```
#include "SpecFileDlg.h"
```

который следует вставить после строки:

```
#include "ex07bView.h"
```

13. **Соберите и протестируйте программу Ex07b.** Щелчок левой кнопки должен открывать диалоговое окно Delete File, в котором вы сможете просматривать каталог и удалять файлы. Осторожно: не уничтожьте каких-нибудь важных файлов!

Прочие возможности адаптации *CFileDialog*

В примере Ex07b вы добавили в диалоговое окно одну кнопку. Столь же несложно добавить и другие элементы управления: просто включите их в шаблон ресурса, и, если это стандартные элементы управления Windows (скажем, поля ввода или списки), вы сможете вставить в свой производный класс переменные-члены и DDX/DDV-код, используя Add Member Variable Wizard. Клиентская программа установит эти переменные-члены перед вызовом функции *DoModal* и получит их обновленные значения после возврата из *DoModal*.

Примечание Даже если вы не применяете вложенных диалоговых окон, все равно объекту *CFileDialog* сопоставлены два окна. Допустим, вы переопределили *OnInitDialog* в производном классе и хотите присвоить какой-нибудь значок диалоговому окну, предназначенному для выбора файлов. Тогда вы должны вызвать *CWnd::GetParent*, чтобы получить окно верхнего уровня по аналогии с тем, что вы делали в программе Ex07b.

```
HICON hIcon = AfxGetApp()->LoadIcon(ID_MYICON);
GetParent()->SetIcon(hIcon, TRUE);     // установка крупного значка
GetParent()->SetIcon(hIcon, FALSE);    // установка мелкого значка
```

Немодальные диалоговые окна

Все диалоговые окна, которые мы пока рассмотрели, были модальными. Теперь нам предстоит познакомиться с немодальными и стандартными диалоговыми окнами для современных версий базового Windows-класса CDialog. В обеих применяется диалоговый ресурс, создаваемый в редакторе ресурсов. Если вы планируете применять немодальное диалоговое окно вместе с объектом «вид», вам придется освоить несколько особых приемов.

Создание немодальных диалоговых окон

Как вы уже знаете, при создании модальных диалоговых окон сначала надо задействовать конструктор *CDialog* с параметром-идентификатором прикрепленного ресурса, чтобы сконструировать объект «диалоговое окно», а потом вывести модальное диалоговое окно на экран, вызвав функцию-член *DoModal*. Окно прекращает свое существование сразу после возврата из *DoModal*. Таким образом, зная, что к тому моменту, когда объект C++ «диалоговое окно» выходит за пределы области видимости, диалоговое окно Windows уже уничтожено, объект «модальное диалоговое окно» можно конструировать на стеке.

Процесс создания немодальных диалоговых окон сложнее. Вы начинаете с вызова конструктора *CDialog* по умолчанию, создавая тем самым объект «диалоговое окно», а вот нужное диалоговое окно создается вызовом функции-члена *CDialog::Create*, а не *DoModal*. *Create* получает идентификатор ресурса как параметр и сразу возвращает управление; при этом диалоговое окно остается на экране. Так что теперь именно вы должны заботиться о том, когда конструировать объект «диалоговое окно», когда создавать само диалоговое окно, когда его уничтожать и когда обрабатывать данные, введенные пользователем.

Различия между созданием модальных и немодальных диалоговых окон таковы (табл. 7-2):

Табл. 7-2. Модальные и немодальные диалоговые окна

	Модальное диалоговое окно	Немодальное диалоговое окно
Используемый конструктор	Конструктор с параметром-идентификатором ресурса	Конструктор по умолчанию (без параметров)
Функция, используемая для создания окна	*DoModal*	*Create* с параметром-идентификатором ресурса

Пользовательские сообщения

Допустим, вы хотите, чтобы немодальное диалоговое окно уничтожалось, когда пользователь щелкает в нем кнопку OK. Сразу же возникает проблема. Как объект «вид» узнает, что пользователь щелкнул кнопку OK? Диалоговое окно могло бы напрямую обратиться к какой-либо функции-члену класса «вид», но это связало бы данное диалоговое окно с конкретным классом «вид». Более удачное решение: диалоговое окно при вызове обработчика кнопки OK отправляет объекту «вид» *пользовательское сообщение* (user-defined message). Получив его, объект «вид» сможет уничтожить диалоговое окно (но не сам объект, что позволит ему сохранить

все введенные в диалоговом окне данные). Итак, вырисовывается сценарий создания нового диалогового окна.

Есть два варианта отправки Windows-сообщений: через функции *CWnd::SendMessage* или *PostMessage*. Первая вызывает функцию-обработчик сообщения сразу, а вторая отправляет сообщение в очередь сообщений Windows. Поскольку *PostMessage* вносит лишь небольшую задержку, можно считать, что функция-обработчик кнопки ОК уже завершилась, когда объект «вид» получает сообщение.

Принадлежность диалогового окна

Теперь предположим, что вы приняли стиль диалогового окна по умолчанию, т. е. диалоговое окно не ограничено клиентской областью окна представления. Поскольку речь идет о Windows, «владелец» диалогового окна — основное окно-рамка приложения (см. главу 12), а не объект «вид». Но надо знать, какой именно объект «вид» сопоставлен диалоговому окну, чтобы отправить этому объекту сообщение. Поэтому ваш класс «диалоговое окно» должен отслеживать свой объект «вид» через переменную-член, которую устанавливает конструктор. Параметр *pParent* конструктора *CDialog* в данном случае не играет никакой роли.

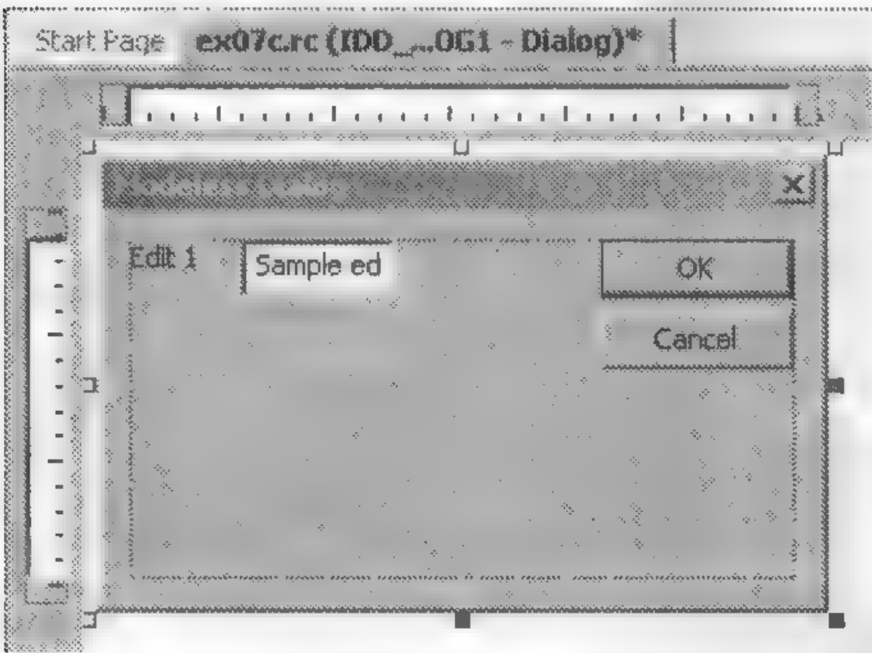
Пример Ex07c: немодальное диалоговое окно

Мы могли бы преобразовать созданное в первой части этой главы диалоговое окно-«монстр» в немодальное, но начать все заново, с простого диалогового окна, пожалуй, легче. В программе-примере Ex07c диалоговое окно оснащено одним полем ввода и кнопками OK и Cancel. Как и в Ex07a, оно открывается щелчком в окне представления, но теперь мы будем удалять его в ответ на другое событие — щелчок *правой* кнопки в окне представления. Мы будем иметь дело только с одним диалоговым окном, поэтому придется позаботиться, чтобы повторное нажатие левой кнопки не приводило к появлению дубликата диалогового окна.

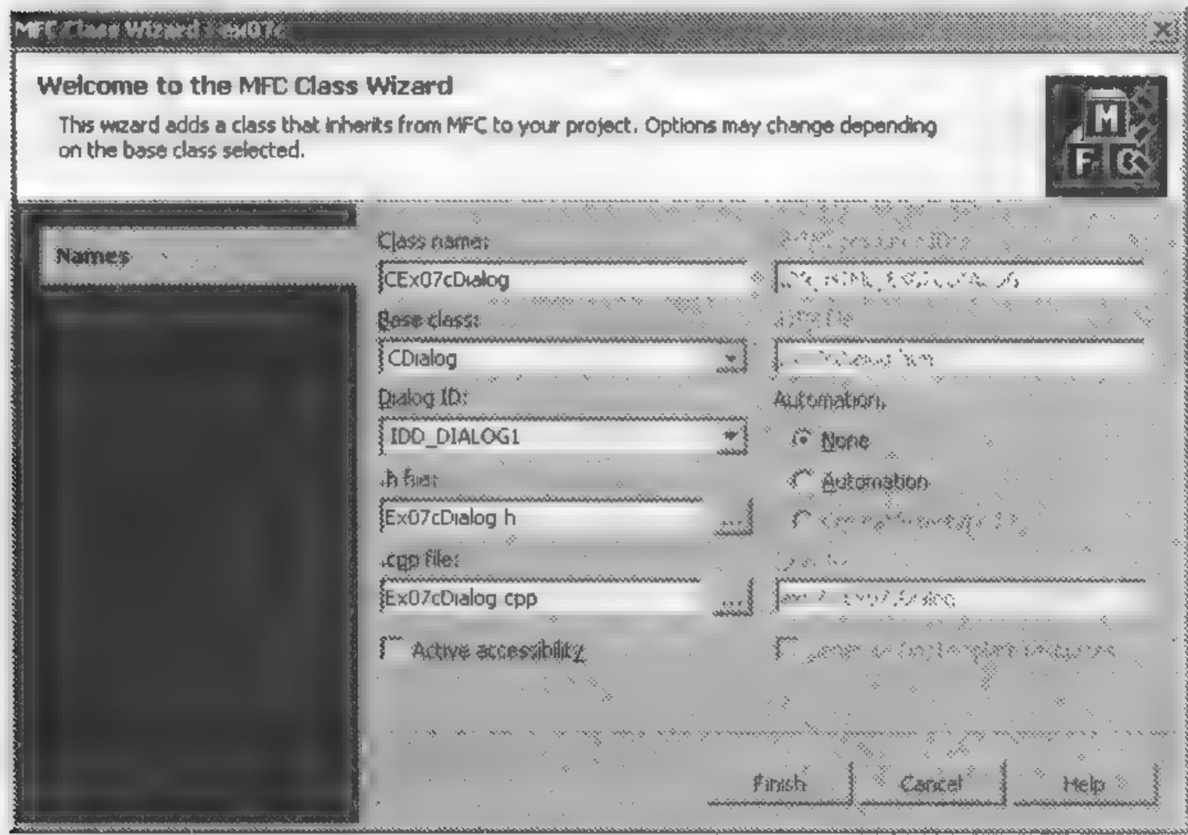
Чтобы кратко объяснить смысл предстоящих операций, скажем, что класс «вид» приложения Ex07c сопоставляется с единственным объектом «диалоговое окно», конструируемым в куче (динамически распределяемой памяти) при конструировании объекта «вид». Диалоговое окно создается и уничтожается в ответ на действия пользователя, но объект «диалоговое окно» уничтожается только по завершении самой программы.

1. **Запустите MFC Application Wizard и создайте проект Ex07c.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **Вызвав редактор диалоговых окон, создайте новый диалоговый ресурс.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс с идентификатором *IDD_DIALOG1*. Измените свойство Caption диалогового окна на Modeless Dialog, а свойство Visible — на TRUE. Оставьте предлагаемые по умолчанию кнопки OK и Cancel с идентификаторами *IDOK* и *IDCANCEL*.

3. **Добавьте элементы управления в диалоговое окно.** Вставьте элемент управления «статический текст» и поле ввода с идентификатором по умолчанию *IDC_EDIT1*. Замените заголовок в элементе управления «статический текст» на Edit 1. У вас должно получиться такое диалоговое окно:



4. **Используя MFC Class Wizard, создайте класс *CEx07cDialog*.** В окне Class View щелкните правой кнопкой проект Ex07c и в контекстном меню последовательно выберите Add и Add Class. В открывшемся диалоговом окне выберите шаблон MFC Class и щелкните кнопку Open, чтобы запустить MFC Class Wizard. Назовите класс *CEx07cDialog*, базовым классом выберите *CDialog*, а в списке Dialog ID — *IDD_DIALOG1*, как показано на рисунке. Закончив настройку, щелкните кнопку Finish.

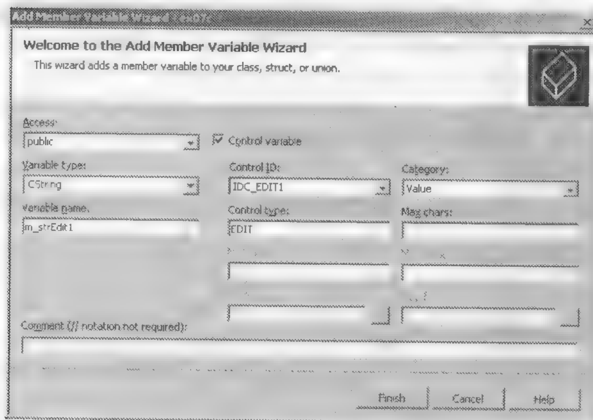


5. **Добавьте в программу перечисленные ниже функции-обработчики сообщений *IDCANCEL* и *IDOK*.** Выберите класс *CEx07cDialog* в Class View и в окне Properties щелкните кнопку Events. Создайте обработчики сообщений *OnBnClickedCancel* и *OnBnClickedOk*:

Идентификатор объекта	Сообщение	Имя функции-члена
IDCANCEL	*BN_CLICKED*	*OnBnClickedCancel*
IDOK	*BN_CLICKED*	*OnBnClickedOk*

6. **Добавьте переменную в класс *CEx07cDialog*.** В окне Class View щелкните класс *CEx07cDialog* и в контекстном меню последовательно выберите Add и Add

Variable. В окне мастера Add Member Variable Wizard добавьте переменную *m_strEdit1* типа *CString* к элементу управления *IDC_EDIT1*:



7. **Отредактируйте Ex07cDialog.h, включив в него указатель на объект «вид» и прототипы функций.** Введите в объявление класса *CEx07cDialog* выделенный код:

```
private:
    CView* m_pView;
```

и добавьте прототипы функций:

```
public:
    CEx07cDialog(CView* pView);
    BOOL Create();
```

Примечание Применяя класс *CView* вместо *CEx07cView*, можно использовать класс «диалоговое окно» с любым классом «вид».

8. **Отредактируйте Ex07cDialog.h, чтобы определить идентификатор сообщения *WM_GOODBYE*.** Добавьте строку:

```
#define WM_GOODBYE WM_USER + 5
```

Windows-константа *WM_USER* — первый из числа доступных идентификаторов для пользовательских сообщений. Некоторыми из них оперирует каркас приложений, поэтому мы пропустим первые пять.

Примечание Visual C++ поддерживает в файле resource.h проекта список символьных определений, но не воспринимает константы, базирующиеся на других константах. Не добавляйте *WM_GOODBYE* в resource.h вручную, потому что Visual C++ может удалить ее.

9. **Добавьте конструктор немодального диалогового окна в файл Ex07cDialog.cpp.** Вы могли бы изменить существующий конструктор *CEx07cDialog*, но, создав отдельный конструктор, вы добьетесь того, что класс «диалоговое

окно» станет пригодным как для модальных, так и для немодальных диалоговых окон. Итак, введите строки:

```
CEx07cDialog::CEx07cDialog(CView* pView)   // "немодальный" конструктор
: m_strEdit1(_T(""))
{
   m_pView = pView;
}
```

Вы должны также вставить в «модальный» конструктор, сгенерированный MFC Application Wizard, строку:

```
IMPLEMENT_DYNAMIC(CEx07cDialog, CDialog)
CEx07cDialog::CEx07cDialog(CWnd* pParent /*=NULL*/)
   : CDialog(CEx07cDialog::IDD, pParent)
   , m_strEdit1(_T(""))
{
   m_pView = NULL;
}
```

Компилятор C++ достаточно «сообразителен», чтобы отличить немодальный конструктор *CEx07cDialog(CView*)* от модального *CEx07cDialog(CWnd*)*. Обнаружив аргумент класса *CView* или производного от него класса *CView*, он генерирует вызов немодального конструктора, а увидев аргумент класса *CWnd* или производного от него класса, — вызов модального конструктора.

10. **Введите в Ex07cDialog.cpp функцию *Create*.** Эта функция производного класса диалогового окна вызывает аналогичную функцию базового класса, передавая идентификатор диалогового ресурса как параметр. Вставьте строки:

```
BOOL CEx07cDialog::Create() {
   return CDialog::Create(CEx07cDialog::IDD);
}
```

Примечание Функция *Create* не является виртуальной. При желании ей можно подобрать и другое имя.

11. **Отредактируйте функции *OnBnClickedCancel* и *OnBnClickedOk* в Ex07cDialog.cpp.** Эти виртуальные функции, сгенерированные мастером Class View, вызываются в ответ на щелчки кнопок в диалоговом окне. Введите выделенный код:

```
void CEx07cDialog::OnBnClickedCancel()
{
   if (m_pView != NULL) {
      // немодальное диалоговое окно - не вызывать OnCancel из базового класса
      m_pView->PostMessage(WM_GOODBYE, IDCANCEL);
   }
   else {
      CDialog::OnCancel(); // если диалоговое окно является модальным
   }
}
```

```
void CEx07cDialog::OnBnClickedOk()
{
    if (m_pView != NULL) {
        // немодальное диалоговое окно - не вызывать OnOK из базового класса
        UpdateData(TRUE);
         m_pView->PostMessage(WM_GOODBYE, IDOK);
    }
    else {
        CDialog::OnOK(); // если диалоговое окно является модальным
    }
}
```

Если диалоговое окно используется как немодальное, оно отсылает объекту «вид» пользовательское сообщение *WM_GOODBYE*. Его обработку мы обсудим позже.

Внимание! Применяя немодальное диалоговое окно, *не вызывайте* функций *CDialog::OnOK* или *CDialog::OnCancel*. Это значит, что вы *обязаны* переопределить эти виртуальные функции в своем производном классе, иначе нажатие клавиш Esc или Enter либо щелчок кнопок мыши приведут к вызову функций базового класса, которые обращаются к Windows-функции *EndDialog*. Последняя подходит только для модальных диалоговых окон. В немодальном диалоговом окне вместо нее нужно вызывать *DestroyWindow*, а чтобы переправить данные от элементов управления диалогового окна переменным-членам класса, вызывайте *UpdateData*.

12. **Отредактируйте заголовочный файл Ex07cView.h.** Для хранения указателя на объект «диалоговое окно» нужна соответствующая переменная-член:

```
private:
    CEx07cDialog* m_pDlg;
```

Если вы добавите в начало файла Ex07cView.h упреждающее объявление:

```
class CEx07cDialog;
```

вам не придется включать Ex07cDialog.h в модули, содержащие Ex07cView.h.

13. **Модифицируйте конструктор и деструктор класса *CEx07cView* в Ex07cView.cpp.** В классе *CEx07cView* есть переменная-член *m_pDlg*, которая указывает на относящийся к представлению объект *CEx07cDialog*. Конструктор объекта «вид» формирует объект «диалоговое окно» в куче, а деструктор объекта «вид» уничтожает последний. Введите выделенный код:

```
CEx07cView::CEx07cView()
{
    m_pDlg = new CEx07cDialog(this);
}

CEx07cView::~CEx07cView()
{
    delete m_pDlg;  // уничтожает окно, если оно еще не уничтожено
}
```

14. **Добавьте код к виртуальной функции *OnDraw* в Ex07cView.cpp.** Функцию *OnDraw* класса *CEx07cView*, шаблон которой создает мастер MFC Application Wizard, нужно запрограммировать так, чтобы она предлагала пользователю щелкнуть кнопкой мыши:

```
void CEx07cView::OnDraw(CDC* pDC)
{
   CEx07cDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDC->TextOut(0, 0, "Press the left mouse button here.")
}
```

15. **Создайте обработчики сообщений мыши *WM_LBUTTONDOWN* и *WM_RBUTTONDOWN* в *CEx07cView*.** Выберите класс *CEx07cView* в Class View, в окне Properties щелкните кнопку Messages, создайте обработчики сообщений *OnLButtonDown* и *OnRButtonDown*, а затем отредактируйте код в файле Ex07cView.cpp:

```
void CEx07cView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // создает диалоговое окно, если оно еще не создано
   if (m_pDlg->GetSafeHwnd() == 0) {
      m_pDlg->Create();  // отображает диалоговое окно на экране
   }
}
void CEx07cView::OnRButtonDown(UINT nFlags, CPoint point)
{
   m_pDlg->DestroyWindow();
   // ничего страшного, если окно уже уничтожено
}
```

Функция *DestroyWindow* не уничтожает объект C++ — это верно для окон почти всех типов, кроме основных окон-рамок. Именно такого ее поведения мы и добиваемся, так как объект «диалоговое окно» мы уничтожаем только в деструкторе объекта «вид».

16. **Дополните файл Ex07cView.cpp оператором #include для заголовочного файла диалогового окна.** Вставьте этот оператор после оператора, включающего заголовочный файл класса «вид»:

```
#include "Ex07cView.h"
#include "Ex07cDialog.h"
```

17. **Напишите код обработки сообщения *WM_GOODBYE*.** Поскольку Class View не поддерживает пользовательские сообщения, этот код придется написать вручную. Здесь-то вы и сможете оценить, какую работу проделывает Visual Studio для других сообщений.

Добавьте в файл Ex07cView.cpp следующую строку между операторами BEGIN_MESSAGE_MAP и END_MESSAGE_MAP:

```
ON_MESSAGE (WM_GOODBYE, OnGoodbye)
```

Кроме того, вставьте в этот файл саму функцию-обработчик сообщения:

```
LRESULT CEx07cView::OnGoodbye(WPARAM wParam, LPARAM lParam)
{
    // сообщение, получаемое в ответ на щелчки кнопок
    // OK и Cancel в немодальном диалоговом окне
    TRACE("CEx07cView::OnGoodbye %x, %lx\n", wParam, lParam);
    TRACE("Dialog edit1 contents = %s\n",
        (const char*) m_pDlg->m_strEdit1);
    m_pDlg->DestroyWindow();
    return 0L;
}
```

В файл Ex07cView.h поместите прототип функции (сразу за прототипами afx_msg функций *OnLButtonDown* и *OnRButtonDown*):

```
afx_msg LRESULT OnGoodbye(WPARAM wParam, LPARAM lParam);
```

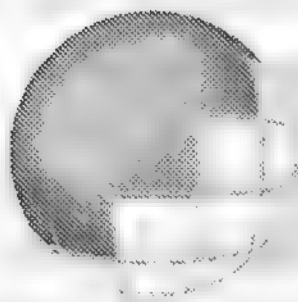
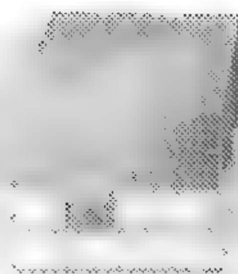
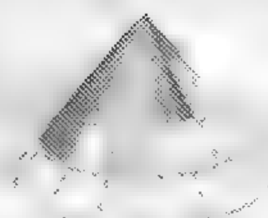
В Win32 параметры *wParam* и *lParam* — обычный путь передачи данных, содержащихся в сообщении. Например, в сообщении о нажатой кнопке мыши в *lParam* упаковываются координаты *x* и *y* курсора мыши. В MFC-библиотеке данные, включаемые в сообщение, передаются через параметры с более внятными именами. Те же координаты курсора передаются как объект *CPoint*. В то же время в пользовательских сообщениях применять *wParam* и *lParam* обязательно, так что в любой момент вы можете задействовать эти две переменные. В этом примере в параметр *wParam* мы записывали идентификатор кнопки.

18. **Соберите и оттестируйте приложение.** Соберите и запустите программу Ex07c. Попробуйте нажать левую кнопку мыши, а затем правую. (В последнем случае курсор мыши должен быть вне диалогового окна.) Вновь вызовите диалоговое окно на экран, введите какие-нибудь данные в элементе управления Edit 1 и щелкните в диалоговом окне кнопку OK. Правильно ли отображает оператор *TRACE* (объекта «вид») содержимое поля ввода?

Примечание При использовании классов «вид» и «диалоговое окно» из программы Ex07c в каком-нибудь MDI-приложении у каждого дочернего окна может быть по одному немодальному диалоговому окну. Закрытие дочернего окна MDI-приложения приводит к тому, что его немодальное диалоговое окно уничтожается, так как деструктор объекта «вид» вызывает деструктор объекта «диалоговое окно».

ГЛАВА

8

Стандартные элементы управления

В главе 7 вы познакомились с такими элементами управления Microsoft Windows, как кнопка (button), флажок (check box), переключатель (radio button), статический текст (static text box), список (list box) и поле со списком (combo box). В этой главе вам предстоит познакомиться еще с одной группой стандартных элементов управления (common control). Код этих элементов управления содержится в файле Windows COMCTL32.DLL, а номер самой последней версии этой библиотеки — 6.0. В ней обновлены существующие и добавлены новые элементы управления. В Microsoft Visual C++ и Microsoft Foundation Class (MFC) существенно улучшена поддержка этих новых элементов управления.

Примечание Версия установленной в системе COMCTL32.DLL определяется версиями ОС Windows и Microsoft Internet Explorer. Эта библиотека имеется в Windows 95, но отсутствует в Windows NT 4.0 — во всех последующих версиях (в том числе Windows 2000/XP) она есть. Она также поставляется в составе Microsoft Internet Explorer версии 3.0 и более поздних.

Для верности и чтобы на иметь неприятностей со «старыми» системами, стоит позаботиться о поставке последней версии COMCTL32.DLL в составе дистрибутива своего приложения. Обновить COMCTL32.DLL можно, установив последнюю версию Internet Explorer. Иногда доступны наборы компонентов с обновленными версиями библиотек, в которые включена COMCTL32.DLL. Самую свежую информацию о COMCTL32.DLL см. в статье «Redistribution of COMCTL32.DLL» (Q186176) справочника *Microsoft Knowledge Base*, а также на сайте компании Microsoft по адресу *http://msdn.microsoft.com*.

Знакомство со стандартными элементами управления

К стандартным элементам управления относятся *индикатор хода процесса* (progress indicator), *ползунок* (slider control), *наборный счетчик* (spin control), *графический* (list control) и *древовидный список* (tree control) (рис. 8-1).

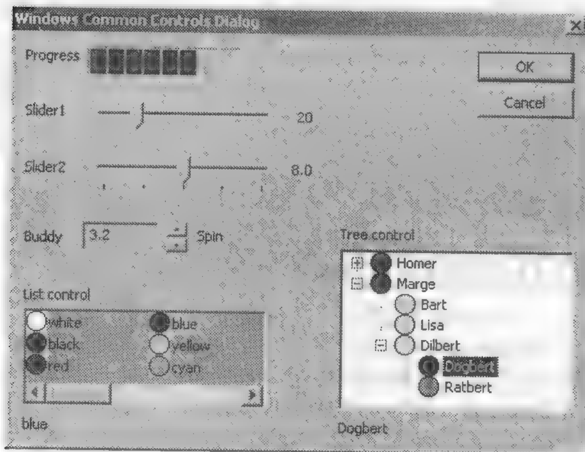


Рис. 8-1. *Стандартные элементы управления Windows*

Элемент управления «индикатор хода процесса»

Индикатор хода процесса — самый простой в программировании стандартный элемент управления; он представлен MFC-классом *CProgressCtrl*. Обычно его используют только для вывода информации. Для инициализации индикатора хода процесса в функции *OnInitDialog* вызываются функции-члены *SetRange* и *SetPos*; после этого при необходимости вызывается *SetPos* из обработчиков сообщений. Значения индикатора хода процесса на рис. 8-1 лежат в диапазоне 0–100 (это диапазон по умолчанию).

Элемент управления «ползунок»

Ползунок (иногда в англоязычной литературе его называют trackbar) представлен классом *CSliderCtrl* и позволяет вводить «аналоговые» значения. (Ползунки были бы, пожалуй, очень уместны в полях Loyality и Reliability примера Ex07a в главе 7.) Если задать для этого элемента управления большой диапазон (например, от 0 до 100 или более), ползунок будет перемещаться очень плавно, а если малый (скажем, от 0 до 5) — скачками. Программно можно создать шкалу, деления которой соответствуют шагу (дискретности) движения ползунка. Именно в таком режиме работает ползунок, позволяющий в окне свойств экрана выбирать текущее разрешение. Диапазон по умолчанию у ползунков отсутствует.

Ползунок программировать легче, чем полосу прокрутки, так как не надо разбирать сообщения *WM_HSCROLL* или *WM_VSCROLL* в классе «диалоговое окно». После установки диапазона ползунок можно перемещать мышью или щелкая деления

шкалы. И все же, если вы хотите отображать значения, связанные с положением ползунка, в другом элементе управления, работать с сообщениями о прокрутке все же придется. Текущее положение ползунка возвращает функция-член *GetPos*. В диалоговом окне на рис. 8-1 верхний ползунок плавно перемещается в диапазоне от 0 до 100, а нижний — дискретно в диапазоне от 0 до 4, и эти индексы связаны со значениями двойной точности (4.0, 5.6, 8.0, 11.0 и 16.0).

Элемент управления «наборный счетчик»

Наборный счетчик (класс *CSpinButtonCtrl*) — это уменьшенный вариант полосы прокрутки, который обычно используется совместно с полем ввода. Поле ввода слева от наборного счетчика (в последовательности переключения между элементами управления по нажатию клавиши Tab оно располагается непосредственно перед счетчиком), иногда называют его *«спутником»* («buddy»). Суть работы данного элемента в том, что пользователь, поместив курсор на счетчик и удерживая левую кнопку мыши, может увеличивать или уменьшать («набирать») число в поле ввода. Скорость «набора» тем выше, чем дольше удерживается кнопка мыши.

Если приложение оперирует в поле ввода с целыми числами, можно вообще обойтись без программирования на C++. Просто закрепите за полем ввода (стандартными средствами Visual Studio) целочисленную переменную-член и не забудьте установить диапазон значений счетчика в функции *OnInitDialog*. (Скорее всего вам не понравится диапазон наборного счетчика по умолчанию: от минимального значения 100 до максимального 0.) Не забудьте установить свойства Auto Buddy и Set Buddy Integer наборного счетчика в TRUE. Для изменения диапазона и схемы ускорения «набора» значений можно вызывать из *OnInitDialog* функции-члены *SetRange* и *SetAccel*.

Чтобы в поле ввода отображались нецелые значения (например, время или числа с плавающей точкой), надо создать обработчик сообщения *WM_VSCROLL* (или *WM_HSCROLL*) от наборного счетчика и предусмотреть в обработчике преобразование целых значений наборного счетчика в числа, отображаемые в поле ввода.

Элемент управления «графический список»

Этот элемент управления (класс *CListCtrl*) вводят, если нужен список, способный отображать не только текст, но и графику. На рис. 8-1 он отображает список с маленькими значками. Элементы располагаются столбцами, поэтому данный элемент управления содержит горизонтальную полосу прокрутки. Когда пользователь выбирает какой-то элемент, элемент управления отправляет уведомляющее сообщение, которое вы должны обработать в своем классе «диалоговое окно». Обработчик сообщения определяет, какой именно элемент выбран. Элементы обозначаются целочисленными индексами, начиная с 0.

И графический, и древовидный списки получают отображаемые ими картинки от стандартного элемента управления — *списка изображений* (image list) (класс *CImageList*). Ваша программа должна формировать список изображений из значков или растровых изображений, а затем передавать элементу управления «графический список» указатель на этот список. Формировать список лучше всего в функции *OnInitDialog*, там же можно определить и нужные текстовые строки. Вставить элемент в список позволяет функция-член *InsertItem*.

Программировать графический список несложно, если ограничиться только строками и значками. Если же вы хотите обеспечить поддержку drag-and-drop или включить в список более сложную, определенную самим пользователем графику, придется потрудиться.

Элемент управления «древовидный список»

Вы уже знакомы с этим элементом управления, если работали с программой Windows Explorer (Проводник) или окном проекта в Solution Explorer среды Visual Studio. MFC-класс *CTreeCtrl* позволяет легко добавить в вашу программу такие же возможности. Древовидный список на рис. 8-1 иллюстрирует состав одной американской семьи. Пользователь может раскрывать и свертывать элементы, щелкая кнопки «+» и «–» или дважды щелкая сами элементы. Значок, расположенный рядом с каждым элементом, программируется так, чтобы вид его изменялся, когда элемент выбран одинарным щелчком.

У графического и древовидного списков есть ряд общих черт: эти списки могут работать с одним и тем же списком изображений и совместно использовать одни и те же уведомляющие сообщения. Однако методы идентификации элементов у них разные. В древовидном списке вместо целочисленных индексов применяется описатель *HTREEITEM*. Для добавления новых элементов служит функция-член *InsertItem*, но перед этим нужно сформировать структуру *TV_INSERTSTRUCT*, которая (помимо прочего) идентифицирует строку, индекс в списке изображений и описатель родительского элемента (*NULL* для элементов верхнего уровня).

Как и графические, древовидные списки предоставляют безграничные возможности настройки. Можно, например, разрешить пользователю редактировать, вставлять и удалять элементы.

Сообщение *WM_NOTIFY*

Раньше элементы управления Windows посылали свои уведомления в сообщениях *WM_COMMAND*. Однако стандартных 32-разрядных параметров *wParam* и *lParam* недостаточно для передачи всей информации из стандартных элементов управления их объектам-родителям. Microsoft решила проблему «пропускной способности», введя новое сообщение *WM_NOTIFY*. При отправке такого сообщения *wParam* содержит идентификатор элемента управления, а *lParam* служит указателем на структуру *NMHDR*, поддерживаемую данным элементом управления. На языке С эта структура определяется так:

```
typedef struct tagNMHDR {
   HWND hwndFrom;  // описатель элемента управления, отправляющего сообщение
   UINT idFrom;    // идентификатор элемента управления
   UINT code;      // особый, характерный для этого элемента код уведомления
} NMHDR;
```

Однако многие элементы управления посылают сообщения *WM_NOTIFY* с указателями на структуры более крупные, чем *NMHDR*. Эти структуры содержат не только показанные выше три поля данных, но и дополнительные, характерные для конкретного элемента управления. Например, во многих уведомлениях из

древовидных списков передается указатель на структуру *NM_TREEVIEW*, содержащую структуру *TV_ITEM* и другие данные. Поэтому, сопоставив обработчику сообщение WM_NOTIFY, Visual Studio генерирует указатель на соответствующую структуру.

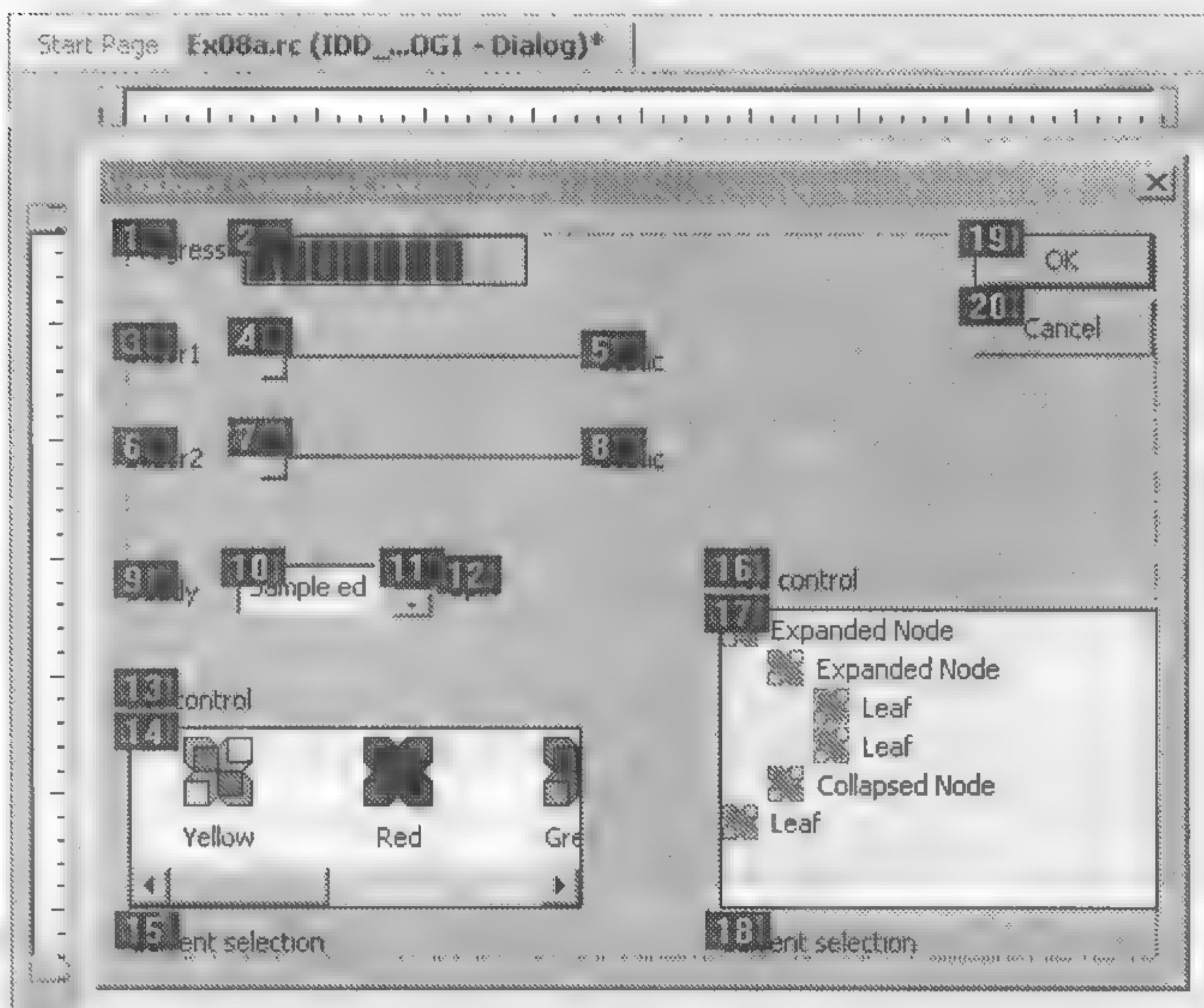
Пример Ex08a: стандартные элементы управления

Мы не станем изобретать реальную программу, в которой используются все элементы управления, — просто включим их в модальное диалоговое окно.

1. **Запустите MFC Application Wizard и создайте проект Ex08a.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex07a. На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения. Закончив настройку, щелкните кнопку Finish.

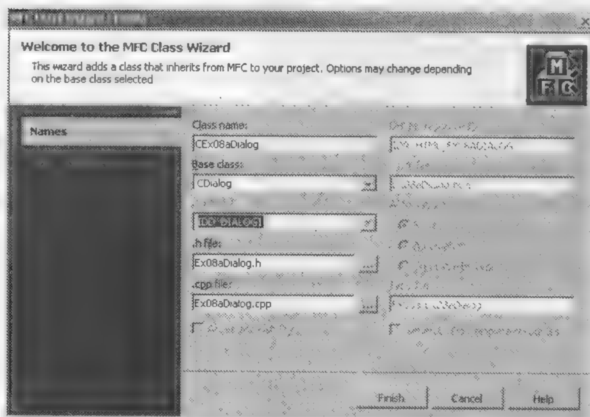
   О других свойствах пока не беспокойтесь — вы установите их на следующих этапах. (Элементы управления могут выглядеть иначе, чем на рис. 8-1, пока вы не установите их свойства.)

2. **Создайте новый диалоговый ресурс с идентификатором *IDD_DIALOG1*.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс Перетащите нужные элементы с панели инструментов Toolbox (если она не видна, выберите в меню View команду Toolbox.) в создаваемое диалоговое окно, а затем разместите их и измените размер. В таблице перечислены типы элементов управления и их идентификаторы. После настройки свойств Caption в диалоговом окне должны быть следующие элементы управления и порядок обхода:



Тип элемента	Идентификатор дочернего окна	Порядок обхода
Статический текст	*IDC_STATIC*	1
Индикатор хода процесса	*IDC_PROGRESS1*	2
Статический текст	*IDC_STATIC*	3
Ползунок	*IDC_SLIDER1*	4
Статический текст	*IDC_STATIC_SLIDER1*	5
Статический текст	*IDC_STATIC*	6
Ползунок	*IDC_SLIDER2*	7
Статический текст	*IDC_STATIC_SLIDER2*	8
Статический текст	*IDC_STATIC*	9
Поле ввода	*IDC_BUDDY_SPIN1*	10
Наборный счетчик	*IDC_SPIN1*	11
Статический текст	*IDC_STATIC*	12
Статический текст	*IDC_STATIC*	13
Графический список	*IDC_LISTVIEW1*	14
Статический текст	*IDC_STATIC_LISTVIEW1*	15
Статический текст	*IDC_STATIC*	16
Древовидный список	*IDC_TREEVIEW1*	17
Статический текст	*IDC_STATIC_TREEVIEW1*	18
Командная кнопка	*IDOK*	19
Командная кнопка	*IDCANCEL*	20

3. **С помощью MFC Class Wizard создайте новый класс *CEx08aDialog*, производный от *CDialog*.** Чтобы запустить MFC Class Wizard, в Class View щелкните класс *CEx08aDialog* правой кнопкой и в контекстном меню последовательно выберите команды Add и Add Class. В качестве базового выберите класс *CDialog*, а в списке Dialog ID выберите *IDD_DIALOG1*:



4. **Переопределите функцию *OnInitDialog* и создайте обработчики сообщений *WM_HSCROLL* и *WM_VSCROLL*.** Для этого в окне Class View выберите класс *CEx08aDialog*, в окне Properties щелкните кнопку Overrides и создайте функцию *OnInitDialog*. В окне Properties щелкните кнопку Messages и создайте

функции-обработчики *OnHScroll* и *OnVScroll* событий *WM_HSCROLL* и *WM_VSCROLL* соответственно.

5. **Запрограммируйте индикатор хода процесса.** Так как Visual Studio не генерирует переменную-член для этого элемента управления, сделайте это вручную. Добавьте в заголовок класса *CEx08aDialog* открытую (public) целочисленную переменную-член *m_nProgress* и присвойте ей в конструкторе значение 0. Кроме того, добавьте в функцию-член *OnInitDialog* код:

```
// Индикатор хода процесса
CProgressCtrl* pProg =
   (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
pProg->SetRange(0, 100);
pProg->SetPos(m_nProgress);
```

6. **Запрограммируйте «непрерывный» ползунок.** Добавьте в заголовок класса *CEx08aDialog* открытую целочисленную переменную-член *m_nTrackbar1* и присвойте ей в конструкторе значение 0. Затем дополните функцию-член *OnInitDialog* следующим кодом, который устанавливает диапазон, а также начальное положение по значению переменной-члена; в соседнем поле статического текста задается число, соответствующее положению ползунка.

```
// Ползунок
CString strText1;
CSliderCtrl* pSlide1 =
   (CSliderCtrl*) GetDlgItem(IDC_SLIDER1);
pSlide1->SetRange(0, 100);
pSlide1->SetPos(m_nSlider1);
strText1.Format("%d", pSlide1->GetPos());
SetDlgItemText(IDC_STATIC_SLIDER1, strText1);
```

Чтобы можно было обновлять статический текст в соответствии с текущим положением ползунка, создайте обработчик сообщения *WM_HSCROLL*, которое ползунок передает объекту «диалоговое окно». Код обработчика выглядит так:

```
void CEx08aDialog::OnHScroll(UINT nSBCode, UINT nPos,
                              CScrollBar* pScrollBar)
{
   CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
   CString strText;
   strText.Format("%d", pSlide->GetPos());
   SetDlgItemText(IDC_STATIC_SLIDER1, strText);
}
```

Наконец, вам надо обновить переменную-член *m_nSlider1*, когда пользователь щелкнет кнопку OK. Так и тянет включить этот код в обработчик кнопки *OnOK*. Но тогда, если в каком-нибудь из других элементов диалогового окна выявится ошибка при проверке корректности введенных данных, вам не избежать проблем. Обработчик установит *m_nSlider1*, даже если пользователь «отменит» диалоговое окно. Поэтому код надо вставить в функцию *DoDataExchange*. И если вы сами проверяете корректность введенных данных, то,

обнаружив ошибку, вызовите *CDataExchange::Fail* — она откроет информационное окно и уведомит пользователя о допущенной им ошибке.

```
void CEx08aDialog::DoDataExchange(CDataExchange* pDX)
{
   if (pDX->m_bSaveAndValidate) {
      TRACE("updating slider data members\n");
      CSliderCtrl* pSlide1 =
      (CSliderCtrl*) GetDlgItem(IDC_SLIDER1);
      m_nSlider1 = pSlide1->GetPos();
   }

   CDialog::DoDataExchange(pDX);
}
```

7. **Запрограммируйте «дискретный» ползунок.** Добавьте в заголовок класса *CEx08aDialog* открытую целочисленную переменную-член *m_nSlider2* и обнулите ее в конструкторе. Эта переменная-член служит индексом для *dValue* — закрытой статической переменной-члена, которая представляет собой массив чисел (4.0, 5.6, 8.0, 11.0 и 16.0). Определите *dValue* в файле Ex08aDialog.h как закрытый статический массив чисел двойной точности:

```
static double dValue[5];
```

а в файл Ex08aDialog.cpp добавьте строку:

```
double CEx08aDialog::dValue[5] = {4.0, 5.6, 8.0, 11.0, 16.0};
```

Потом дополните функцию-член *OnInitDialog* кодом, определяющим диапазон ползунка, разметку шкалы и начальное положение:

```
CString strText2;
CSliderCtrl* pSlide2 =
   (CSliderCtrl*) GetDlgItem(IDC_SLIDER2);
pSlide2->SetRange(0, 4);
pSlide2->SetPos(m_nSlider2);
strText2.Format("%3.1f", dValue[pSlide2->GetPos()]);
SetDlgItemText(IDC_STATIC_SLIDER2, strText2);
```

Если б был только один ползунок, обработчик сообщения *WM_HSCROLL*, о котором говорилось на шаге 5, вполне мог бы сработать. Но в программе два ползунка посылают одно и то же сообщение *WM_HSCROLL*, и обработчик должен разобраться в них. Код следует изменить так:

```
void CEx08aDialog::OnHScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar)
{
   CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
   CString strText;

      // два ползунка посылают сообщения
      // HSCROLL (нужна разная обработка)
   switch(pScrollBar->GetDlgCtrlID()) {
   case IDC_SLIDER1:
```

```
        strText.Format("%d", pSlide->GetPos());
        SetDlgItemText(IDC_STATIC_SLIDER1, strText);
        break;
    case IDC_SLIDER2:
        strText.Format("%3.1f", dValue[pSlide->GetPos()]);
        SetDlgItemText(IDC_STATIC_SLIDER2, strText);
        break;
        }
}
```

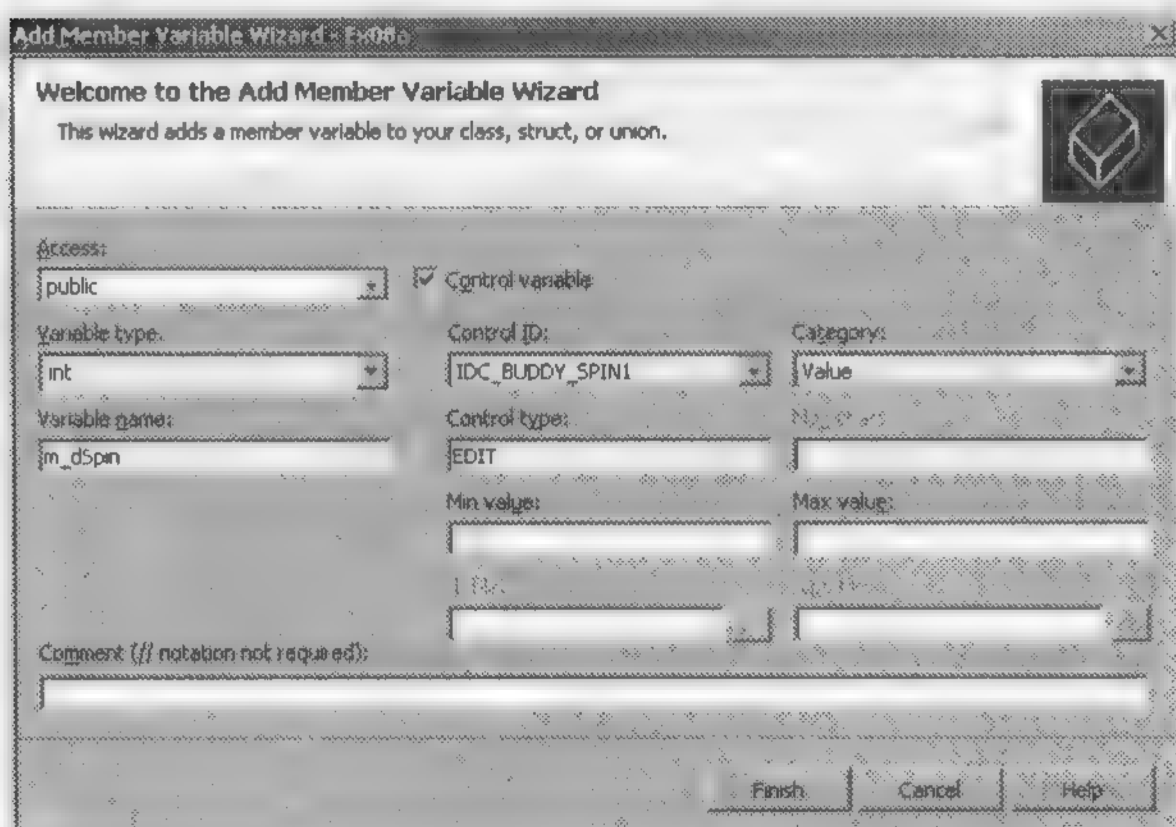
В ползунке Slider2 нужна шкала с делениями, поэтому установите свойства Tick Marks и Auto Ticks этого элемента управления в TRUE. Последний параметр заставит ползунок пометить делениями каждое приращение. Если деления не видны, увеличьте высоту элемента управления.

Для этого ползунка справедливы те же рассуждения по поводу проверки данных, что и для предыдущего. Поэтому вставьте в функцию-член *DoDataExchange* класса «диалоговое окно» в if-блоке, созданном на предыдущем шаге, такой код:

```
CSliderCtrl* pSlide2 =
    (CSliderCtrl*) GetDlgItem(IDC_SLIDER2);
m_nSlider2 = pSlide2->GetPos();
```

Через редактор диалоговых окон задайте значение Bottom/Right свойству Point обоих ползунков, а свойству Align Text элементов управления *IDC_TRACKBAR1* и *IDC_TRACKBAR2* — значение Right.

8. **Запрограммируйте наборный счетчик.** Наборный счетчик связан со своим спутником — полем ввода, всегда предшествующим счетчику при перемещении между элементами управления клавишей Tab. Средствами Add Member Variable Wizard добавьте переменную-член двойной точности *m_dSpin* для поля ввода *IDC_BUDDY_SPIN1*. Мы используем тип *double* вместо *int* только из-за того, что последний практически не требует программирования и задача была бы чересчур проста. Мы хотим установить диапазон поля ввода от 0.0 до 10.0, но сам счетчик оперирует только с целыми числами. Чтобы открыть окно мастера Add Member Variable Wizard, в Class View выберите класс *CEx08aDialog*, а затем — команду Add Variable из меню Project. Настройте свойства класса так:



Добавьте следующий код в *OnInitDialog*, чтобы установить диапазон его значений от 0 до 100 и присвоить ему начальное значение *m_dSpin * 10.0*:

```
// Наборный счетчик
CSpinButtonCtrl* pSpin = (CSpinButtonCtrl*) GetDlgItem(IDC_SPIN1);
pSpin->SetRange(0, 100);
pSpin->SetPos((int) (m_dSpin * 10.0));
```

Чтобы текущее значение счетчика отражалось в связанном с ним поле ввода, нужно предусмотреть обработчик сообщения *WM_VSCROLL*, которое счетчик отправляет в объект «диалоговое окно». Вот этот код:

```
void CEx08aDialog::OnVScroll(UINT nSBCode, UINT nPos,
                CScrollBar* pScrollBar)
{
    if (nSBCode == SB_ENDSCROLL) {
        return; // Отбрасываем ненужные сообщения
    }
    // Обрабатываем сообщения о перемещениях только от IDC_SPIN1
    if (pScrollBar->GetDlgCtrlID() == IDC_SPIN1) {
        CString strValue;
        strValue.Format("%3.1f", (double) nPos / 10.0);
        ((CSpinButtonCtrl*) pScrollBar)->GetBuddy()
                ->SetWindowText(strValue);
    }

    CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

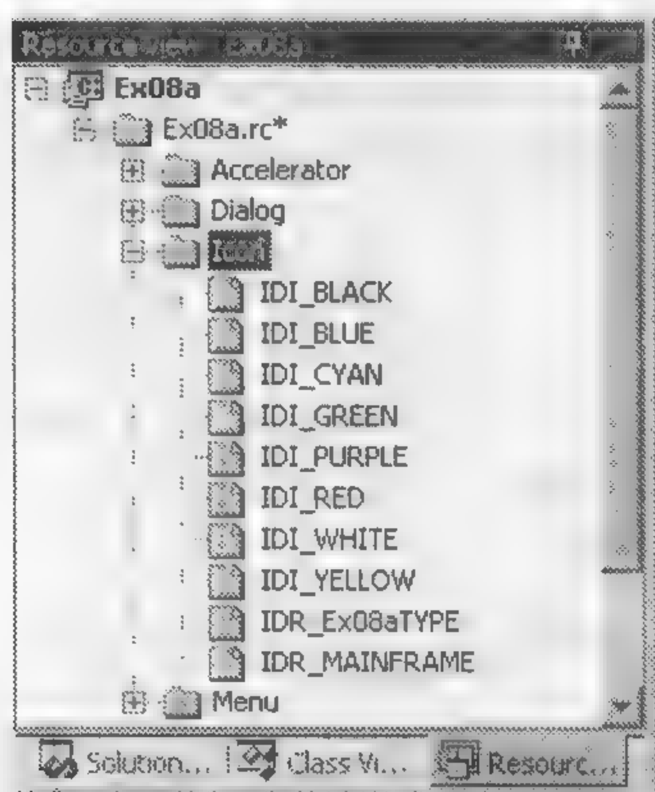
Вводить какой-то код в *OnOK* или *DoDataExchange* не нужно, так как содержимое поля ввода обрабатывает DDX-код (Dialog Data Exchange), отвечающий за обмен данными в диалоговом окне. В редакторе диалоговых окон задайте значение TRUE свойству Auto Buddy наборного счетчика и свойству Read-only связанного со счетчиком поля ввода.

9. **Подготовьте список изображений.** Этот список необходим и графическому, и древовидному списку, а в самом списке нужно создать *значки* (icons). Значки вы найдете в каталоге Ex08a\res на компакт-диске — разноцветные кружки с черной границей. Если у вас есть значки поинтереснее, используйте их.

   Прежде чем импортировать значки в проект Ex08a, скопируйте их в папку Ex08a\res проекта. Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните кнопку Import. В диалогом окне Import перейдите в папку с файлами значков. В поле со списком Files of type выберите тип файлов — Icon Files, выберите файлы с Icon0.ico по Icon7.ico и щелкните Open. Значки откроются в редакторе изображений и появятся в папке Icon в окне Resource View. В окне Properties определите идентификаторы (ID) значков и закройте редактор значков.

Файл значка	Идентификатор ресурса
Icon0.ico	*IDI_WHITE*
Icon1.ico	*IDI_BLACK*
Icon2.ico	*IDI_RED*
Icon3.ico	*IDI_BLUE*
Icon4.ico	*IDI_YELLOW*
Icon5.ico	*IDI_CYAN*
Icon6.ico	*IDI_PURPLE*
Icon7.ico	*IDI_GREEN*

По завершении этой операции папка Icon в Resource View должна выглядеть так:



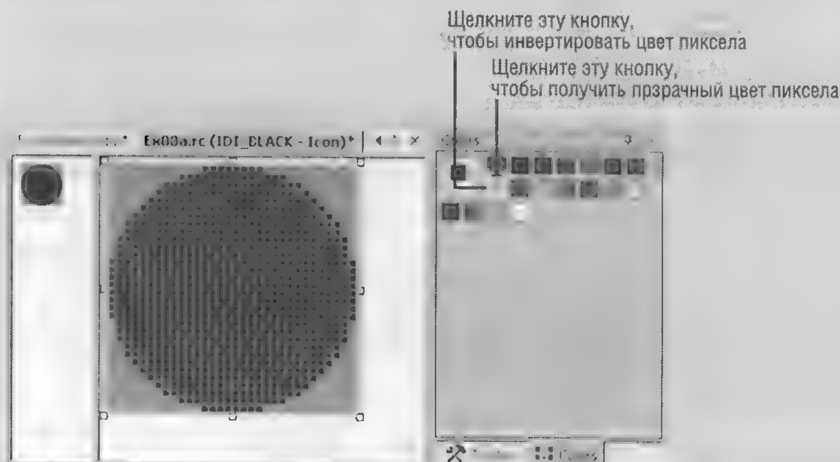
Теперь добавьте в заголовочный файл класса *CEx08aDialog* открытую переменную-член *m_imageList* класса *CImageList* и включите в *OnInitDialog* такой код:

```
// Icons
HICON hIcon[8];
int n;
m_imageList.Create(16, 16, 0, 8, 8); // или 32 - для больших значков
hIcon[0] = AfxGetApp()->LoadIcon(IDI_WHITE);
hIcon[1] = AfxGetApp()->LoadIcon(IDI_BLACK);
hIcon[2] = AfxGetApp()->LoadIcon(IDI_RED);
hIcon[3] = AfxGetApp()->LoadIcon(IDI_BLUE);
hIcon[4] = AfxGetApp()->LoadIcon(IDI_YELLOW);
hIcon[5] = AfxGetApp()->LoadIcon(IDI_CYAN);
hIcon[6] = AfxGetApp()->LoadIcon(IDI_PURPLE);
hIcon[7] = AfxGetApp()->LoadIcon(IDI_GREEN);
for (n = 0; n < 8; n++) {
   m_imageList.Add(hIcon[n]);
}
```

Несколько слов о значках

Вероятно, вы знаете, что растровое изображение, или растр, — это массив битов, представляющих пикселы на экране. (О растровых изображениях мы говорили в главе 6.) В Windows значок — это целый «пучок» растров. Прежде всего надо понимать, что размеры у значка зависят от растровых изображений. Для представления мелкого значка применяется изображение 16×16 пикселов, а для крупного — 32×32. Для каждого размера существует два отдельных растра: один (4 бита/пиксел) для цветного изображения и, как маска, один монохромный (1 бит/пиксел). Если бит маски равен 0, соответствующий пиксел изображения представляет непрозрачный цвет. Если же бит маски равен 1, то при черном (0) цвете изображения данный пиксел прозрачен, а при белом (0xF) — цвет фона в позиции этого пиксела инвертируется.

Маленькие значки появились в Windows 95. Они используются на панели задач (taskbar), в Windows Explorer и в уже упомянутых элементах управления, если программисты предусматривали их. Если у значка нет растрового изображения для размера 16×16 пикселов, Windows сформирует уменьшенный вариант (маленький значок) из растра для размера 32×32 пиксела, но тогда он уже не получится таким аккуратным, как специально нарисованный в разрешении 16×16. Графический редактор позволяет создавать и редактировать значки. Вот как выглядит его палитра цветов:



Верхний квадратик в верхнем левом углу окна палитры показывает основной цвет для кистей, заливки фигур и пр., а квадратик пониже указывает на цвет границ фигур. Основной цвет выбирается щелчком левой кнопки, а цвет границ — щелчком правой. Теперь взгляните на центр верхней части окна палитры. Щелчок верхнего значка монитора позволяет рисовать прозрачные пикселы (изображаются темно-голубым цветом), а нижнего — инвертированные пикселы (изображаются красным цветом).

10. **Запрограммируйте графический список.** В редакторе диалоговых окон установите свойства графического списка, как показано в таблице.

Свойство элемента управления	Значение
Alignment	*Top*
Always Show Selection	*True*
Single Selection	*True*
View	*List*

Затем введите в *OnInitDialog* код:

```
// Графический список
static char* color[] = {"white", "black", "red",
                        "blue", "yellow", "cyan",
                        "purple", "green"};
CListCtrl* pList =
   (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
pList->SetImageList(&m_imageList, LVSIL_SMALL);
for (n = 0; n < 8; n++) {
   pList->InsertItem(n, color[n], n);
}
pList->SetBkColor(RGB(0, 255, 255)); // Тихий  ужас!!
pList->SetTextBkColor(RGB(0, 255, 255));
```

Как показывает последняя строка, со стандартными элементами управления сообщение *WM_CTLCOLOR* не используется — функция вызывается для установки цвета фона Вы просто вызываете специальную функцию. Однако, как вы увидите, запустив программу, инвертированные пикселы значков выглядят весьма убого.

Если вы создадите обработчик уведомляющего сообщения *LVN_ITEMCHANGED* из графического списка, то сможете отслеживать выбор элементов пользователем. В окне Class View выберите класс *CEx08aDialog*, в окне Properties щелкните кнопку Events, разверните элемент *IDC_LISTVIEW1*, выберите событие *LVN_ITEMCHANGED* и добавьте функцию-обработчик *OnLvnItemchangedListview1*. Добавьте в нее следующий код для отображения текста выбранных элементов списка в статическом элементе управления (метке):

```
void CEx08aDialog::OnLvnItemchangedListview1(NMHDR* pNMHDR,
                                             LRESULT* pResult)
{
   LPNMLISTVIEW pNMLV = reinterpret_cast<LPNMLISTVIEW>(pNMHDR);
   CListCtrl* pList =
      (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
   int nSelected = pNMLV->iItem;
   if (nSelected >= 0) {
      CString strItem = pList->GetItemText(nSelected, 0);
      SetDlgItemText(IDC_STATIC_LISTVIEW1, strItem);
   }
   *pResult = 0;
}
```

В структуре *NM_LISTVIEW* есть переменная-член *iItem*, содержащая индекс выбранного элемента списка.

11. **Запрограммируйте древовидный список.** В редакторе диалоговых окон установите атрибуты стиля элемента управления «древовидный список»:

Свойство элемента управления	Значение
Has Buttons	*True*
Has Lines	*True*
Lines At Root	*True*
Scroll	*True*

Затем вставьте в *OnInitDialog* строки:

```
// Древовидный список
CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
pTree->SetImageList(&m_imageList, TVSIL_NORMAL);
//  значения, общие для всей древовидной структуры
TV_INSERTSTRUCT tvinsert;
tvinsert.hParent = NULL;
tvinsert.hInsertAfter = TVI_LAST;
tvinsert.item.mask = TVIF_IMAGE | TVIF_SELECTEDIMAGE |
                     TVIF_TEXT;
tvinsert.item.hItem = NULL;
tvinsert.item.state = 0;
tvinsert.item.stateMask = 0;
tvinsert.item.cchTextMax = 6;
tvinsert.item.iSelectedImage = 1;
tvinsert.item.cChildren = 0;
tvinsert.item.lParam = 0;
// верхний уровеньl
tvinsert.item.pszText = "Homer";
tvinsert.item.iImage = 2;
HTREEITEM hDad = pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Marge";
HTREEITEM hMom = pTree->InsertItem(&tvinsert);
tvinsert.hParent = hDad;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 3;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
// второй уровень
tvinsert.hParent = hMom;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 4;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Dilbert";
HTREEITEM hOther = pTree->InsertItem(&tvinsert);
```

```
// третий уровень
tvinsert.hParent = hOther;
tvinsert.item.pszText = "Dogbert";
tvinsert.item.iImage = 7;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Ratbert";
pTree->InsertItem(&tvinsert);
```

Как видите, этот код устанавливает в *TV_INSERTSTRUCT* текст и индексы изображений, а затем вызывает *InsertItem* для создания узлов в древовидной структуре.

И, наконец, создайте обработчик уведомления *TVN_SELCHANGED* от элемента управления «древовидный список». В окне Class View выберите класс *CEx08aDialog*, в окне Properties щелкните кнопку Events, разверните элемент *IDC_TREEVIEW1*, выберите событие *TVN_SELCHANGED* и добавьте функцию-обработчик *OnTvnSelchangedTreeview1*. Добавьте выделенный код для отображения выбранного текста в статическом элементе управления (метке):

```
void CEx08aDialog::OnTvnSelchangedTreeview1 (NMHDR* pNMHDR,
                                             LRESULT* pResult)
{
   LPNMTREEVIEW pNMTreeView = reinterpret_cast<LPNMTREEVIEW>(pNMHDR;
   CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
   HTREEITEM hSelected = pNMTreeView->itemNew.hItem;
   if (hSelected != NULL) {
      char text[31];
      TV_ITEM item;
      item.mask = TVIF_HANDLE | TVIF_TEXT;
      item.hItem = hSelected;
      item.pszText = text;
      item.cchTextMax = 30;
      VERIFY(pTree->GetItem(&item));
      SetDlgItemText(IDC_STATIC_TREEVIEW1, text);
   }
   *pResult = 0;
}
```

В структуре *NM_TREEVIEW* есть переменная-член *itemNew*, содержащая информацию о выбранном узле, а точнее его описатель. Функция *GetItem* отыскивает данные, относящиеся к этому узлу, и получает текст, используя указатель, хранящийся в структуре *TV_ITEM*. Переменная *mask* сообщает Windows, что описатель *hItem* достоверен и что нужно вернуть текст.

12. **Отредактируйте виртуальную функцию *OnDraw* в файле Ex08aView.cpp.** Выделенный код заменяет существующий:

```
void CEx08aView::OnDraw(CDC* pDC)
{
   CEx08aDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

13. **Добавьте функцию-член *OnLButtonDown*.** В окне Class View выберите класс *CEx08aDialog*, в окне Properties щелкните кнопку Messages, выберите сообщение *WM_LBUTTONDOWN* и добавьте функцию-обработчик *OnLButtonDown*. Отредактируйте код, как показано ниже:

```
void CEx08aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CEx08aDialog dlg;

    dlg.m_nSlider1 = 20;
    dlg.m_nSlider2 = 2;    // индекс для 8.0
    dlg.m_nProgress = 70;  // только запись
    dlg.m_dSpin = 3.2;

    dlg.DoModal();

    CView::OnLButtonDown(nFlags, point);
}
```

Чтобы включить файл Ex08aDialog.h в файл Ex08aView.cpp, добавьте оператор:

```
#include "Ex08aDialog.h"
```

14. **Скомпилируйте и запустите программу.** Поэкспериментируйте с элементами управления, чтобы увидеть, как они работают. Индикатор хода процесса мы с вами не запрограммировали — о нем поговорим в главе 13.

Дополнительные стандартные элементы управления Windows

К дополнительным стандартным элементам управления Windows относятся элемент выбора даты и времени (date and time picker), календарь на месяц (month calendar), элемент ввода адреса Интернета (internet protocol address control) и расширенное поле со списком (extended combo box control). Все они используются в примере Ex08b. Диалоговое окно Ex08b показано на рис. 8-2, к которому вам не раз придется обращаться в процессе работы.

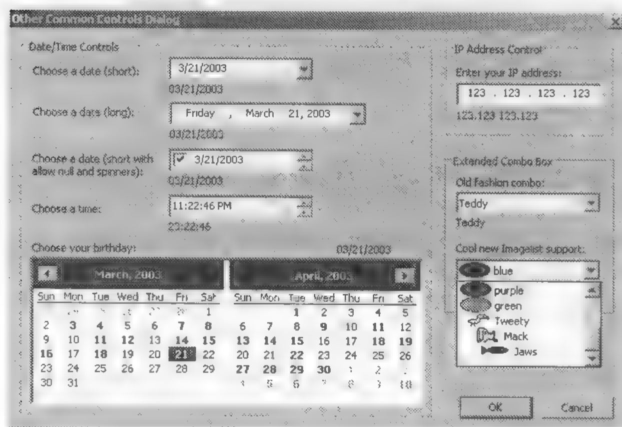


Рис. 8-2. *Дополнительные стандартные элементы управления в диалоговом окне*

Элемент выбора даты и времени

Место для ввода даты и времени — типовое поле в диалоговых окнах. До того, как с выходом элементов управления IE4 появился стандартный *элемент выбора даты и времени* (date and time picker), разработчикам приходилось использовать элементы управления сторонних поставщиков либо создавать подкласс MFC-класса «поле ввода», где выполнялась значительная по объему проверка правильности введенной даты и времени. К счастью, новый элемент управления предоставляет пользователю массу возможностей при выборе даты и времени, а разработчику — обширную гамму стилей и параметров. Так, даты можно отображать в коротком (14.8.68) или длинном (14 августа 1968) форматах. Время можно вводить в хорошо знакомом пользователю формате «часы-минуты-секунды» с 12- или 24-часовым периодом.

Данный элемент управления также позволяет вам решить, где пользователь будет указывать дату: в поле ввода, в календаре или в наборном счетчике. Среди других характеристик отметим возможность одиночного или множественного (для указания диапазона дат) выбора и возможность включить/отключить обведение текущей даты красным кружком. Есть даже режим, позволяющий установить флажок «нет даты». На рис. 8-2 первые четыре элемента управления с левой стороны иллюстрируют различные конфигурации элемента выбора даты и времени.

В MFC класс *CDateTimeCtrl* предоставляет MFC-интерфейс к этому элементу управления. Класс поддерживает несколько уведомлений, расширяющих возможности программирования элемента управления. Кроме того, в нем есть функции-члены для работы со структурами *CTime* и *COleDateTime*.

Для установки даты и времени в *CDateTimeCtrl* служит функция *SetTime*, а для их получения — *GetTime*. Функция *SetFormat* позволяет задать собственные форматы отображения.

Классы *CTime* и *COleDateTime*

Большинство программистов, знакомых с MFC, привыкло использовать класс *CTime*. Однако, поскольку допустимый для него диапазон дат лежит между 1 января 1970 г. и 18 января 2038 г., многим понадобилась альтернатива. Одна из них — класс *COleDateTime*, который поддерживает Automation OLE и обрабатывает даты между 1 января 100 г. и 31 декабря 9999 г. У каждого из классов свои плюсы и минусы. Так, *CTime* прекрасно справляется со всеми особенностями перехода на летнее время, а *COleDateTime* — нет.

COleDateTime часто выбирают из-за его более широкого диапазона. Любое приложение, в котором применяется *CTime*, в ближайшие 40 лет придется переработать — ведь он не поддерживает даты, следующие за 2038 г. Выбор класса определяется конкретными нуждами и ожидаемым временем жизни приложения.

Календарь на месяц

Большое окно слева внизу на рис. 8-2 — это *календарь на месяц* (month calendar). Подобно элементу выбора даты и времени, этот элемент управления позволяет выбрать дату. Однако его можно задействовать и для реализации небольшого *лич-*

ного ежедневника (Personal Information Manager, PIM). Вы можете одновременно вывести столько месяцев, на сколько хватит места — от одного месяца до нескольких лет. В примере Ex08b календарь показывает только два месяца.

Данный элемент управления поддерживает одиночный и множественный выборы, а также позволяет выбрать различные параметры отображения, например, нумерацию месяцев или обведение кружочком текущего дня. Уведомления позволяют разработчику указать, какие даты выделить полужирным. Выбор того, что должны означать даты, выделенные полужирным начертанием, целиком и полностью предоставлен разработчику. Например, вы можете обозначить таким образом праздники, назначенные встречи или свободные дни. В MFC этот элемент управления реализован в классе *CMonthCalCtrl*.

Для инициализации класса *CMonthCalCtrl* вызывается функция-член *SetToday*. Функции этого класса, в том числе и *SetToday*, могут работать как с *CTime*, так и с *COleDateTime*.

Элемент ввода IP-адреса

При разработке приложения, использующего в том или ином виде Интернет или TCP/IP, иногда требуется запросить у пользователя IP-адрес. В состав стандартных элементов управления входит *элемент ввода IP-адреса* (IP address control), показанный на рис. 8-2 справа вверху. Кроме получения от пользователя IP-адреса, он автоматически проверяет корректность адреса. Поддержку элемента ввода IP-адреса со стороны MFC обеспечивает класс *CIPAddressCtrl*.

Данный элемент управления состоит из четырех «полей», которые нумеруются слева направо (рис. 8-3).

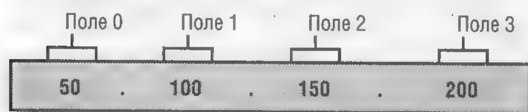


Рис. 8-3. *Поля элемента ввода IP-адреса*

Этот элемент управления инициализируется вызовом функции-члена *SetAddress* в функции *OnInitDialog* вашего диалогового окна. *SetAddress* получает в качестве параметра значение типа DWORD, в котором каждый байт соответствует одному полю. В обработчиках сообщений можно вызвать *GetAddress* для получения как DWORD, так и отдельных байтов, представляющих различные поля IP-адреса.

Расширенное поле со списком

«Старомодное» поле со списком было разработано на заре Windows. Его «возраст» и негибкий дизайн создавали множество проблем. При выпуске элементов управления Microsoft решила включить в их число гораздо более гибкую версию поля со списком — *расширенное поле со списком* (extended combo box).

Этот элемент управления облегчает разработчику доступ к полю редактирования поля со списком и обеспечивает больший контроль над ним. Кроме того, к элементам списка можно присоединить *список изображений* (image list). Вывод графических изображений в новом элементе управления довольно прост, особенно в сравнении с ранее применявшимися полями со списком с *программной прори-*

совкой (owner-drawn). Каждому элементу списка можно сопоставить три типа изображения: *выделенное* (selected image), *невыделенное* (unselected image) и *перекрытое* (overlay image). Они позволяют получить различные виды вывода, что вы и увидите в примере Ex08b. Два расширенных поля со списком показаны внизу на рис. 8-2. MFC-класс *CComboBoxEx* полностью поддерживает расширенное поле со списком.

Подобно «графическому списку» (см. выше), *CComboBoxEx* можно присоединить к *CImageList*, который в дальнейшем автоматически выводит графические изображения около текста в расширенном поле со списком. Если вы уже знакомы с *CComboBox*, то *CComboBoxEx* может вызвать замешательство: вместо строк в нем хранятся элементы типа *COMBOBOXEXITEM* — структуры, которая состоит из следующих полей.

- ***UINT mask*** — набор битовых флагов, определяющих, какие операции выполняются при помощи этой структуры. Например, если в результате операции устанавливается или должно быть получено поле изображения, устанавливается флаг *CBEIF_IMAGE*.
- ***INT_PTR iItem*** — номер элемента. Подобно обычному полю со списком, в расширенном элементе управления нумерация начинается с 0.
- ***LPSTR pszText*** — текст элемента.
- ***int cchTextMax*** — длина буфера, на который указывает *pszText*[1].
- ***int iImage*** — порядковый номер (начиная с нуля) в присоединенном списке изображений.
- ***int iSelectedImage*** — порядковый номер (начиная с 0) в присоединенном списке изображений, который используется для представления «выбранного» состояния.
- ***int iOverlay*** — порядковый номер (начиная с 0) в присоединенном списке изображений, который служит для перекрытия текущего изображения.
- ***int iIndent*** — число 10-пиксельных отступов.
- ***LPARAM lParam*** — 32-разрядный произвольный параметр, связанный с элементом.

Вы увидите, как использовать эту структуру, в примере Ex08b.

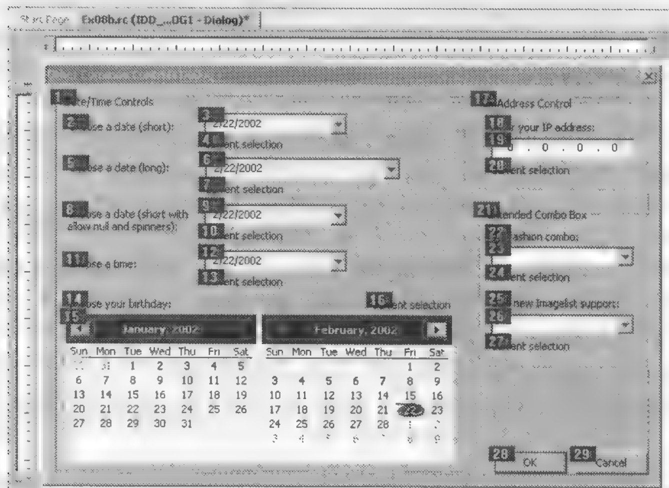
Пример Ex08b: Дополнительные стандартные элементы управления

Мы построим диалоговое окно, в котором создадим и запрограммируем дополнительные стандартные элементы управления всех типов.

1. **Запустите MFC Application Wizard и создайте проект Ex08b.** Выберите в меню File последовательно команды New и Project. В диалоговом окне New Project качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex08b. На странице Application Type мастера установите переключатель в положение Single document, оставив остальные параметры без изменения.

[1] Только при наличии текста элемента. — *Прим. перев.*

2. **Создайте новый ресурс диалогового окна с идентификатором *IDD_DIALOG1*.** Выберите в меню Project среды разработки команду Add Resource и создайте новый диалоговый ресурс. Разместите элементы управления в диалоговом окне с помощью окна Toolbox. (Если его не видно, выберите в меню View команду Toolbox.) В таблице показан список элементов управления, их идентификаторы и порядок обхода. После определения свойств Caption статического текста в диалоговом окне должны быть следующие элементы управления и порядок обхода.



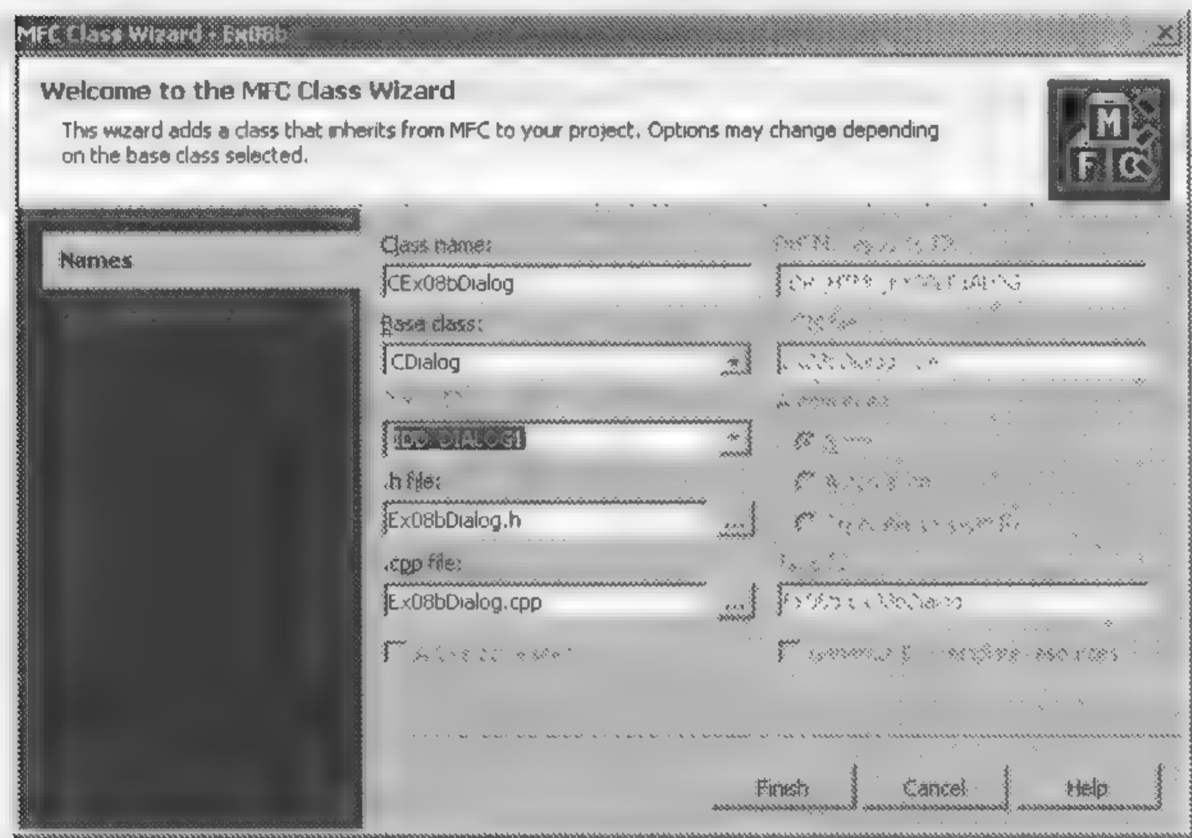
Однако пока не установлены некоторые свойства, наше диалоговое окно не будет выглядеть в точности, как на рис. 8-2.

Тип элемента управления	Идентификатор дочернего окна	Последовательность при нажатии Tab
Группирующая рамка	*IDC_STATIC*	1
Статический текст	*IDC_STATIC*	2
Элемент выбора даты/времени	*IDC_DATETIMEPICKER1*	3
Статический текст	*IDC_STATIC1*	4
Статический текст	*IDC_STATIC*	5
Элемент выбора даты/времени	*IDC_DATETIMEPICKER2*	6
Статический текст	*IDC_STATIC2*	7
Статический текст	*IDC_STATIC*	8
Элемент выбора даты/времени	*IDC_DATETIMEPICKER3*	9
Статический текст	*IDC_STATIC3*	10
Статический текст	*IDC_STATIC*	11
Элемент выбора даты/времени	*IDC_DATETIMEPICKER4*	12
Статический текст	*IDC_STATIC4*	13
Статический текст	*IDC_STATIC*	14
Календарь на месяц	*IDC_MONTHCALENDAR1*	15
Статический текст	*IDC_STATIC5*	16
Группирующая рамка	*IDC_STATIC*	17

см. след. стр.

Тип элемента управления	Идентификатор дочернего окна	Последовательность при нажатии Tab
Статический текст	*IDC_STATIC*	18
Элемент ввода адреса Интернета	*IDC_IPADDRESS1*	19
Статический текст	*IDC_STATIC6*	20
Группирующая рамка	*IDC_STATIC*	21
Статический текст	*IDC_STATIC*	22
Расширенное поле со списком	*IDC_COMBOBOXEX1*	23
Статический текст	*IDC_STATIC7*	24
Статический текст	*IDC_STATIC*	25
Расширенное поле со списком	*IDC_COMBOBOXEX2*	26
Статический текст	*IDC_STATIC8*	27
Кнопка	*IDOK*	28
Кнопка	*IDCANCEL*	29

3. **Используйте MFC Class Wizard для создания нового класса *CEx08bDialog*, производного от *CDialog*.** Выберите класс в Class View, а затем —команду Add Class в меню Project. В окне MFC Class Wizard в качестве базового выберите класс *CDialog*, а в поле со списком Dialog ID — *IDD_DIALOG1*:



Переопределите функцию *OnInitDialog*. Выберите класс *CEx08bDialog* в Class View. Щелкните кнопку Overrides в верней части окна Properties и добавьте функцию *OnInitDialog*.

4. **Настройте свойства элементов управления в диалоговом окне.** Чтобы продемонстрировать все разнообразие элементов управления, нам нужно установить свойства каждого из стандартных элементов управления.

   - **Элемент выбора даты и времени в коротком формате.** В первом элементе выбора даты и времени (*IDC_DATETIMEPICKER1*) убедитесь, что установлен короткий формат даты (по умолчанию): свойству Format задано значение Short Date.
   - **Элемент выбора даты и времени в длинном формате.** Второй элемент выбора даты и времени (*IDC_DATETIMEPICKER2*) настройте на длинный формат, задав значение Long Date свойству Format.

- **Элемент выбора даты и времени в коротком формате с дополнительными возможностями.** В третьем элементе выбора даты и времени (*IDC_DATETIMEPICKER3*) задайте значение Short Date свойству Format, а свойствам Allow Edit, Show None и Use Spin Control — TRUE.
- **Элемент выбора времени.** Четвертый элемент выбора даты и времени (*IDC_DATETIMEPICKER4*) настройте для выбора времени, а не даты. Для этого свойству Format присвойте значениеTime, а свойству Use Spin Control — TRUE.
- **Календарь на месяц.** Сначала свойству Day States задайте TRUE. Кроме того, по умолчанию календарь в диалоговом окне вообще выглядит не так, как элемент управления — у него нет обрамления. Чтобы он был похож на другие, установите свойства Client Edge и Static Edge в TRUE.
- **Элемент ввода IP-адреса.** У этого элемента управления (*IDC_IPADDRESS1*) никаких свойств менять не нужно.
- **Расширенные поля со списком.** У этих элементов управления (*IDC_COMBOBOXEX1* и *IDC_COMBOBOXEX2*) свойств менять не нужно.

5. **Добавьте переменные в класс *CEx08bDialog*.** Запустите Add Member Variable Wizard, выбрав класс *CEx08bDialog* в окне Class View, а затем щелкнув команду Add Variable в меню Project. Для каждого элемента управления создайте соответствующую переменную из таблицы.

Идентификатор	Категория	Тип	Имя переменной
IDC_DATETIMEPICKER1	Элемент управления (Control)	*CDateTimeCtrl*	*m_MonthCal1*
IDC_DATETIMEPICKER2	Элемент управления (Control)	*CDateTimeCtrl*	*m_MonthCal2*
IDC_DATETIMEPICKER3	Элемент управления (Control)	*CDateTimeCtrl*	*m_MonthCal3*
IDC_DATETIMEPICKER4	Элемент управления (Control)	*CDateTimeCtrl*	*m_MonthCal4*
IDC_IPADDRESS1	Элемент управления (Control)	*CIPAddressCtrl*	*m_ptrIPCtrl*
IDC_MONTHCALENDAR1	Элемент управления (Control)	*CMonthCalCtrl*	*m_MonthCal5*
IDC_STATIC1	Переменная (Value)	*CString*	*m_strDate1*
IDC_STATIC2	Переменная (Value)	*CString*	*m_strDate2*
IDC_STATIC3	Переменная (Value)	*CString*	*m_strDate3*
IDC_STATIC4	Переменная (Value)	*CString*	*m_strDate4*
IDC_STATIC5	Переменная (Value)	*CString*	*m_strDate5*
IDC_STATIC6	Переменная (Value)	*CString*	*m_strIPValue*
IDC_STATIC7	Переменная (Value)	*CString*	*m_strComboEx1*
IDC_STATIC8	Переменная (Value)	*CString*	*m_strComboEx2*

6. **Запрограммируйте элемент выбора даты/времени в сокращенном формате.** В данном примере нас не интересует начальная дата, отображаемая в элементе управления, поэтому мы не обрабатываем его в *OnInitDialog*. Если же необходимо задать дату, придется в *OnInitDialog* вызвать метод *SetTime*. Кроме того, когда во время выполнения пользователь выбирает новую дату, связан-

ный статический элемент управления должен автоматически обновляться. Для этого нужно позаботиться об обработчике сообщения *DTN_DATETIMECHANGE*. Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Events в окне Properties, разверните узел *IDC_DATETIMEPICKER1*, выберите сообщение *DTN_DATETIMECHANGE* и создайте обработчик *OnDtnDatetimechangeDatetimepicker1*. Затем добавьте в обработчик для элемента управления Datetimepicker1 следующий код:

```
void CEx08bDialog::OnDtnDatetimechangeDatetimepicker1 (NMHDR* pNMHDR,
   LRESULT* pResult)
{
   LPNMDATETIMECHANGE pDTChange =
      reinterpret_cast<LPNMDATETIMECHANGE>(pNMHDR);
   CTime ct;
   m_MonthCal1.GetTime(ct);
   m_strDate1.Format(_T("%02d/%02d/%2d"),
                  ct.GetMonth(),ct.GetDay(),ct.GetYear());
   UpdateData(FALSE);
   *pResult = 0;
}
```

В этом коде для записи значения времени в переменную *ct* типа *CTime* применяется переменная-член *m_MonthCal1*, соответствующая первому элементу выбора даты/времени. Затем функцией *CString::Format* устанавливается текст соответствующего статического элемента управления. Наконец, с помощью *UpdateData(FALSE)* запускается DDX MFC и вызывается автоматическое обновление статического элемента управления.

7. **Запрограммируйте элемент выбора даты/времени в «длинном» формате.** Создадим аналогичный обработчик для второго элемента выбора даты/времени.

```
void CEx08bDialog::OnDtnDatetimechangeDatetimepicker2(NMHDR* pNMHDR,
   LRESULT* pResult)
{
   LPNMDATETIMECHANGE pDTChange =
      reinterpret_cast<LPNMDATETIMECHANGE>(pNMHDR);
   CTime ct;
   m_MonthCal2.GetTime(ct);
   m_strDate2.Format(_T("%02d/%02d/%2d"),
                  ct.GetMonth(),ct.GetDay(),ct.GetYear());
   UpdateData(FALSE);
   *pResult = 0;
}
```

8. **Запрограммируйте третий элемент выбора даты/времени.** Для этого нужен аналогичный обработчик, но, поскольку мы установили в его свойствах флаг Show None, пользователь может задать «недействительную» (NULL) дату, щелкнув встроенный флажок. Вместо вызова *GetTime* «вслепую» мы должны проверить возвращаемое значение. Если оно не нулевое, то пользователь выбрал NULL-дату, в противном случае — допустимую дату. Как и в двух предыдущих обработчиках, значение объекта *CTime* преобразуется в строку и автоматически отображается в соответствующем статическом элементе управления:

```
void CEx08bDialog::OnDtnDatetimechangeDatetimepicker3(NMHDR* pNMHDR,
   LRESULT* pResult)
{
   LPNMDATETIMECHANGE pDTChange =
      reinterpret_cast<LPNMDATETIMECHANGE>(pNMHDR);
   // ВНИМАНИЕ: может иметь значение NULL!
   CTime ct;
   int nRetVal = m_MonthCal3.GetTime(ct);
   if (nRetVal) // Если это не 0, то null;
                // а если это так, делайте то, что следует.
   {
      m_strDate3 = "NO DATE SPECIFIED!!"; // Дата не указана
   }
   else
   {
      m_strDate3.Format(_T("%02d/%02d/%2d"),ct.GetMonth(),
                  ct.GetDay(),ct.GetYear());
   }
   UpdateData(FALSE);
   *pResult = 0;
}
```

9. **Запрограммируйте элемент ввода времени.** Для элемента ввода времени нужен аналогичный обработчик, но формат вывода вместо «месяцы/дни/годы» будет «часы/минуты/секунды»:

```
void CEx08bDialog::OnDtnDatetimechangeDatetimepicker4(NMHDR* pNMHDR,
   LRESULT* pResult)
{
   LPNMDATETIMECHANGE pDTChange =
      reinterpret_cast<LPNMDATETIMECHANGE>(pNMHDR);
   CTime ct;
   m_MonthCal4.GetTime(ct);
   m_strDate4.Format(_T("%02d:%02d:%2d"),
               ct.GetHour(),ct.GetMinute(),ct.GetSecond());
   UpdateData(FALSE);
   *pResult = 0;
}
```

10. **Запрограммируйте календарь на месяц.** Обработчик для календаря похож на обработчик для элемента выбора даты/времени, но между ними есть и различия. Во-первых, чтобы определить смену даты пользователем, нужно обрабатывать сообщение *MCN_SELCHANGE*. Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Events в окне Properties, разверните узел *IDC_MONTHCALENDER1*, выберите сообщение *MCN_SELCHANGE* и создайте обработчик *OnMcnSelchangeMonthcalender1*. Помимо другого имени обработчика, в этом элементе управления применяется функция *GetCurSel*, а не *GetTime*. Вот обработчик *MCN_SELCHANGE* для элемента управления «календарь на месяц»:

```
void CEx08bDialog::OnMcnSelchangeMonthcalendar1(NMHDR* pNMHDR,
   LRESULT* pResult)
{
```

```
    LPNSELCHANGE pSelChange =
        reinterpret_cast<LPNMSELCHANGE>(pNMHDR);
    CTime ct;
    m_MonthCal5.GetCurSel(ct);
    m_strDate5.Format(_T("%02d/%02d/%2d"),
                ct.GetMonth(),ct.GetDay(),ct.GetYear());
    UpdateData(FALSE);
    *pResult = 0;
}
```

11. **Запрограммируйте элемент ввода IP-адреса.** Сначала нужно убедиться, что элемент управления инициализирован. Для этого мы передаем ему значение 0 типа DWORD. Если не выполнить инициализацию, все без исключения поля элемента управления будут пустыми. Добавьте в *CDialog1::OnInitDialog* вызов:

```
// Инициализация  элемента ввода адреса Интернета
m_ptrIPCtrl.SetAddress(0L);
```

Теперь нам нужен обработчик сообщения для обновления связанного статического элемента управления при всяком изменении IP-адреса. Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Events в окне Properties, разверните узел *IDC_IPADDRESS1*, выберите сообщение *IPN_FIELDCHANGED* и создайте обработчик *OnIpnFieldchangedIpaddress1*.

А затем реализуем обработчик следующим образом:

```
void CEx08bDialog::OnIpnFieldchangedIpaddress1(NMHDR* pNMHDR,
    LRESULT* pResult)
{
    LPNMIPADDRESS pIPAddr =
        reinterpret_case<LPNMIADDRESS>(pNMHDR);
    DWORD dwIPAddress;
    m_ptrIPCtrl.GetAddress(dwIPAddress);

    m_strIPValue.Format("%d.%d.%d.%d   %x.%x.%x.%x",
        HIBYTE(HIWORD(dwIPAddress)),
        LOBYTE(HIWORD(dwIPAddress)),
        HIBYTE(LOWORD(dwIPAddress)),
        LOBYTE(LOWORD(dwIPAddress)),
        HIBYTE(HIWORD(dwIPAddress)),
        LOBYTE(HIWORD(dwIPAddress)),
        HIBYTE(LOWORD(dwIPAddress)),
        LOBYTE(LOWORD(dwIPAddress)));
    UpdateData(FALSE);
    *pResult = 0;
}
```

Вызов *CIPAddressCtrl::GetAddress* помещает IP-адрес в локальную переменную *dwIPAddress* типа *DWORD*. Далее следует довольно сложный вызов *CString::Format*, разделяющий *DWORD* на отдельные поля. Здесь применяется макрос *LOWORD* для получения младшего слова, макрос *HIWORD* для получения старшего слова и макросы *HIBYTE/LOBYTE* для выделения отдельных полей.

12. **Добавьте элементы списка для первого расширенного поля со списком.** Допишите следующий код, чтобы добавить программно элементы «George» «Sandy» и «Teddy» в первое расширенное поле со списком. Заметили ли вы разницу между этим и обычным полем со списком?

```
// Инициализация IDC_COMBOBOXEX1
CComboBoxEx* pCombo1 =
   (CComboBoxEx*) GetDlgItem(IDC_COMBOBOXEX1);
CString rgstrTemp1[3];
rgstrTemp1[0] = "George";
rgstrTemp1[1] = "Sandy";
rgstrTemp1[2] = "Teddy";

COMBOBOXEXITEM cbi1;
cbi1.mask = CBEIF_TEXT;
for (int nCount = 0; nCount < 3; nCount++)
{
   cbi1.iItem = nCount;
   cbi1.pszText = (LPTSTR)(LPCTSTR)rgstrTemp1[nCount];
   cbi1.cchTextMax = 256;
   pCombo1->InsertItem(&cbi1);
}
```

Первое, что бросается в глаза, — использование структуры *COMBOBOXEXITEM* вместо простых целых чисел в предыдущих версиях этого элемента управления.

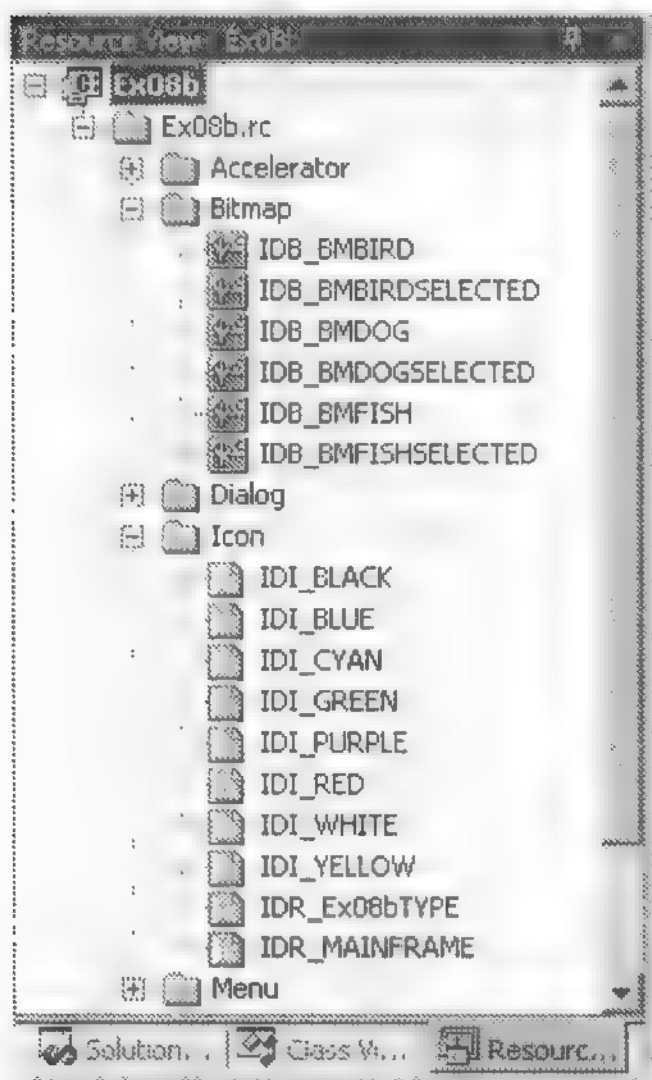
13. **Добавьте обработчик для первого расширенного поля со списком.** Нам нужен обработчик сообщения *CBN_SELCHANGE* от первого поля со списком. Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Events в окне Properties, разверните узел *IDC_COMBOBOX1*, выберите сообщение *CBN_SELCHANGE* и создайте обработчик *OnCbnSelchangeComboboxex1*. Вот измененный код обработчика поля со списком.

```
void CEx08bDialog::OnCbnSelchangeComboboxex1 ()
{
   COMBOBOXEXITEM cbi;
   CString str ("dummy_string"); // просто строка
   CComboBoxEx * pCombo = (CComboBoxEx *)GetDlgItem(IDC_COMBOBOXEX1);

   int nSel = pCombo->GetCurSel();
   cbi.iItem = nSel;
   cbi.pszText = (LPTSTR)(LPCTSTR)str;
   cbi.mask = CBEIF_TEXT;
   cbi.cchTextMax = str.GetLength();
   pCombo->GetItem(&cbi);
   SetDlgItemText(IDC_STATIC7,str);
   return;
}
```

Получив описание элемента, обработчик извлекает строку и вызывает *SetDlgItemText* для обновления связанного статического элемента управления.

14. **Добавьте изображения к элементам второго расширенного поля со списком.** Первое поле со списком не требует специального программирования. Оно просто демонстрирует, как реализовать простое, похожее на «обычное» расширенное поле со списком. Но для второго поля со списком потребуется довольно много программировать. Сначала нужно раздобыть 6 растровых изображений и 8 значков и добавить их в ресурсы проекта. Вы вправе взять их с компакт-диска или любые другие, имеющиеся под рукой. Добавьте эти растры и значки в Resource View и задайте их идентификаторы, как показано на рисунке.



Прежде чем добавлять изображения к расширенному полю со списком, создадим в классе *CEx08bDialog* открытую переменную-член *m_imageList* типа *CImageList*. Добавьте в заголовочный файл Ex08bDialog.h строку:

```
CImageList m_imageList;
```

Теперь мы можем добавить к списку изображений несколько растров и затем «связать» изображения с тремя элементами, добавленным в расширенное поле со списком. Добавьте в метод *OnInitDialog* класса *CEx08bDialog* такой код:

```
// Инициализация IDC_COMBOBOXEX2
CComboBoxEx* pCombo2 =
   (CComboBoxEx*) GetDlgItem(IDC_COMBOBOXEX2);
// Сначала добавим изображения к элементам.
// У нас 6 растров, которые мы сопоставим нашим строкам:

m_imageList.Create(32,16,ILC_MASK,12,4);

CBitmap bitmap;

bitmap.LoadBitmap(IDB_BMBIRD);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();
```

```
bitmap.LoadBitmap(IDB_BMBIRDSELECTED);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();

bitmap.LoadBitmap(IDB_BMDOG);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();

bitmap.LoadBitmap(IDB_BMDOGSELECTED);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();

bitmap.LoadBitmap(IDB_BMFISH);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();

bitmap.LoadBitmap(IDB_BMFISHSELECTED);
m_imageList.Add(&bitmap, (COLORREF)0xFFFFFF);
bitmap.DeleteObject();

// Определяем список изображений
pCombo2->SetImageList(&m_imageList);

CString rgstrTemp2[3];
rgstrTemp2[0] = "Tweety";
rgstrTemp2[1] = "Mack";
rgstrTemp2[2] = "Jaws";

COMBOBOXEXITEM cbi2;
cbi2.mask = CBEIF_TEXT|CBEIF_IMAGE|CBEIF_SELECTEDIMAGE|CBEIF_INDENT;
int nBitmapCount = 0;
for (int nCount = 0; nCount < 3; nCount++)
{
    cbi2.iItem = nCount;
    cbi2.pszText = (LPTSTR)(LPCTSTR)rgstrTemp2[nCount];
    cbi2.cchTextMax = 256;
    cbi2.iImage = nBitmapCount++;
    cbi2.iSelectedImage = nBitmapCount++;
    cbi2.iIndent = (nCount & 0x03);
    pCombo2->InsertItem(&cbi2);
}
```

Сначала, используя *GetDlgItem*, получаем указатель на элемент управления. Затем вызываем *Create* для выделения памяти под изображения и инициализации списка изображений. Следующая серия вызовов загружает каждое растровое изображение, добавляет его в список изображений и удаляет ресурс, выделенный при загрузке.

Для привязки *m_imageList* к расширенному полю со списком вызывается *CComboBoxEx::SetImageList*. Далее инициализируется структура *COMBOBOXEXITEM*, и в цикле *for* от 0 до 2 для каждого элемента устанавливаются изображения в выбранном и невыбранном состоянии. Создается массив строк *rgstrTemp*,

связанный с каждым изображением и состоящий из трех элементов: «Tweety», «Mack» и «Jaws». Переменная *nBitmapCount* служит для настройки строки поля со списком, она также задает идентификатор растрового изображения, помещаемый в структуру *COMBOBOXEXITEM*. В теле цикла сначала вызывается *CComboBoxEx::GetItem*, чтобы получить сведения о каждом элементе расширенного поля со списком. Затем для элемента задаются растровые изображения, и вызывается *CComboBoxEx::SetItem* для передачи измененной структуры *COMBOBOXEXITEM* обратно в список и завершения связывания изображений с существующими элементами списка.

15. **Добавьте элементы во второй расширенное поле со списком.** Другой способ размещения изображений в расширенном поле со списком — добавить их динамически, как показано в коде, добавленном в *OnInitDialog*:

```
HICON hIcon[8];
int n;
// Теперь добавим несколько цветных значков
hIcon[0] = AfxGetApp()->LoadIcon(IDI_WHITE);
hIcon[1] = AfxGetApp()->LoadIcon(IDI_BLACK);
hIcon[2] = AfxGetApp()->LoadIcon(IDI_RED);
hIcon[3] = AfxGetApp()->LoadIcon(IDI_BLUE);
hIcon[4] = AfxGetApp()->LoadIcon(IDI_YELLOW);
hIcon[5] = AfxGetApp()->LoadIcon(IDI_CYAN);
hIcon[6] = AfxGetApp()->LoadIcon(IDI_PURPLE);
hIcon[7] = AfxGetApp()->LoadIcon(IDI_GREEN);
for (n = 0; n < 8; n++) {
   m_imageList.Add(hIcon[n]);
}

static char* color[] = {"white", "black", "red",
                        "blue", "yellow", "cyan",
                        "purple", "green"};

cbi2.mask = CBEIF_IMAGE|CBEIF_TEXT|CBEIF_OVERLAY|
         CBEIF_SELECTEDIMAGE;

for (n = 0; n < 8; n++) {
   cbi2.iItem = n;
   cbi2.pszText = color[n];
   cbi2.iImage = n+6;         // 6 - это смещение в списке изображений
   cbi2.iSelectedImage = n+6; // на 6 уже добавленных элементов...
   cbi2.iOverlay = n+6;
   int nItem = pCombo2->InsertItem(&cbi2);
   ASSERT(nItem == n);
}
```

Добавление значков в вышеприведенном коде напоминает пример Ex08a. В цикле *for* заполняется структура *COMBOBOXEXITEM*, а затем вызывается *CComboBoxEx::InsertItem* для добавления отдельных элементов.

16. **Добавьте обработчик для второго расширенного поля со списком.** Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Events в окне Properties,

разверните узел *IDC_COMBOBOX2*, выберите сообщение *CBN_SELCHANGE* и создайте обработчик *OnCbnSelchangeComboboxex2*. Добавьте выделенный код — это в сущности тот же обработчик, что и для первого поля со списком :

```
void CEx08bDialog::OnCbnSelchangeComboboxex2()
{
    COMBOBOXEXITEM cbi;
    CString str ("dummy_string"); // просто строка
    CComboBoxEx * pCombo = (CComboBoxEx *)GetDlgItem(IDC_COMBOBOXEX2);
    int nSel = pCombo->GetCurSel();
    cbi.iItem = nSel;
    cbi.pszText = (LPTSTR)(LPCTSTR)str;
    cbi.mask = CBEIF_TEXT;
    cbi.cchTextMax = str.GetLength();
    pCombo->GetItem(&cbi);
    SetDlgItemText(IDC_STATIC8,str);
    return;
}
```

17. **Свяжите диалоговое окно с классом «вид».** Добавьте следующий код в виртуальную функцию *OnDraw* в файле Ex08bView.cpp. Выделенный код заменяет имеющийся:

```
void CEx08bView::OnDraw(CDC* pDC)
{
    CEx08vDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

18. **Добавьте функцию-член *OnLButtonDown* к классу *CEx08bView*.** Выберите класс *CEx08bDialog* в Class View, щелкните кнопку Messages в окне Properties, выберите сообщение *WM_LBUTTONDOWN* и создайте обработчик *OnLButtonDown*. Отредактируйте код таким образом:

```
void CEx08aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CEx08bDialog dlg;
    dlg.DoModal();

    CView::OnLButtonDown(nFlags, point);
}
```

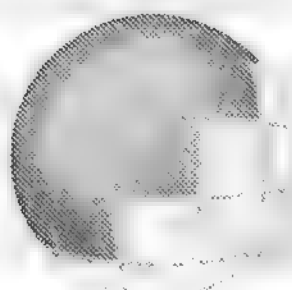
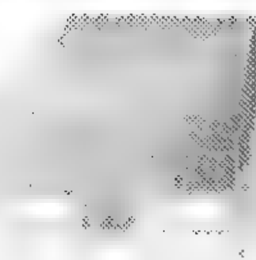
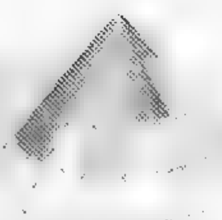
Добавьте следующий оператор в файл Ex08bView.cpp, чтобы включить в него заголовочный файл Ex08bDialog.h:

```
#include "Ex08bDialog.h"
```

19. **Соберите и запустите программу.** Теперь вы можете поэкспериментировать со стандартными элементами управления, чтобы увидеть их в работе и понять, насколько они годятся для ваших приложений.

ГЛАВА

9

Использование элементов управления ActiveX

Разработчики всегда искали пути «компонентизации» элементов пользовательского интерфейса. Хотя в Microsoft Windows много встроенных элементов управления, таких как кнопки или поля ввода, было бы неплохо иметь и другие элементы управления, скажем, диаграммы или таблицы. В «классической» разработке в среде Windows эта проблема решается созданием элементов управления ActiveX (раньше их называли OLE controls или OCX). Элементы управления ActiveX доступны как в Microsoft Visual Basic, так и в Microsoft Visual C++.

Даже с появлением Microsoft .NET элементы ActiveX не стали менее важны. В этой главе речь пойдет о применении ActiveX-элементов в приложениях Visual C++ .NET. Предполагается, что вы сможете научиться их использовать, даже не имея особых знаний о лежащей в их основе модели COM, — в конце концов Microsoft не требует, чтобы программисты на VB были экспертами по COM. Но, чтобы уметь писать элементы управления ActiveX, вам потребуется немного больше, в первую очередь знание фундаментальных основ COM. В главах 22–28 мы детально расскажем о COM, а также о применении ATL для создания элементов управления ActiveX. Если вы всерьез настроены *создавать* такие элементы управления, отсылаем вас к книге Адама Деннинга (Adam Denning) «ActiveX Controls Inside Out» (Microsoft Press, 1997). Но, конечно, осведомленность в каких-то вопросах теории элементов управления ActiveX не повредит, когда вы будете использовать их в своих программах. Основы, с которых можно начать, вы найдете в этой книге в главах 24, 25 и 30. Даже в мире Microsoft .NET технологии ActiveX найдется место для создания гибких компонентных прикладных пользовательских интерфейсов.

ActiveX и обычные элементы управления Windows

Элемент управления ActiveX — это программный модуль, подключаемый к программе так же, как и элемент управления Windows. По крайней мере так кажется. Сравним элементы ActiveX и уже известные нам элементы управления Windows.

Основные характеристики обычных элементов управления

В главе 8 мы применяли обычные элементы управления Windows, такие как поля ввода и списки, и познакомились со стандартными элементами управления Windows, которые работают в целом аналогично. Все они представляют собой дочерние окна и применяются главным образом в диалоговых окнах — для их представления в MFC служат классы, такие как *CEdit* и *CTreeCtrl*. За создание дочернего окна элемента управления всегда отвечает клиентская программа.

Обычные элементы управления посылают диалоговому окну уведомления (стандартные Windows-сообщения), например *BN_CLICKED*. Если вы хотите что-то сделать с элементом управления, вы вызываете функцию-член соответствующего класса C++, которая передает этому элементу Windows-сообщение. Все элементы управления по своей природе — окна. MFC-классы для элементов управления являются производными от класса *CWnd*, поэтому, чтобы получить текст из поля ввода, вы обращаетесь к *CWnd::GetWindowText*. Но работа и этой функции строится на передаче сообщения элементу управления.

Элементы управления Windows — неотъемлемая часть этой ОС, хотя стандартные элементы управления Windows и находятся в отдельном DLL-модуле. Кроме того, есть еще одна разновидность обычных элементов управления Windows — *пользовательские элементы управления* (custom controls). (Их создает сам программист, и они не являются частью ОС, но работают как обычные элементы управления, т. е. передают родительскому окну уведомления *WM_COMMAND* и обрабатывают посылаемые ему сообщения.) Один из пользовательских элементов управления описан в главе 22.

Общие черты ActiveX- и обычных элементов управления

ActiveX-элемент можно рассматривать как дочернее окно, подобно обычному элементу управления. В диалоговое окно его вставляет редактор диалоговых окон, после чего его идентификатор появляется в шаблоне ресурса. При динамическом создании элемента ActiveX («на лету») вызывается функция *Create* класса, представляющего данный элемент управления. Обычно это делается в обработчике сообщения *WM_CREATE* родительского окна. Чтобы выполнить какие-либо операции с элементами управления ActiveX, вызывается функция-член C++ (так же, как и для элемента управления Windows). Окно, содержащее элементы ActiveX, называется *контейнером* (container).

Различия ActiveX- и обычных элементов управления

Наиболее очевидная отличительная черта элементов управления ActiveX — наличие у них свойств и методов. Все функции C++, которые вы вызываете для работы с ними, так или иначе используют методы и свойства. Свойства имеют символические имена, которым соответствуют целочисленные индексы. (Эти индексы

называются DISPID, и мы познакомимся с ними в главе 23.) Разработчик элемента ActiveX присваивает каждому свойству определенное имя, скажем, *BackColor* или *GridLineWidth*, и тип, например, строка, целое или целое двойной точности. Для растровых изображений и значков существует даже такой тип свойства, как *картинка* (Picture). Клиентская программа может устанавливать отдельные свойства элемента управления, задавая их целочисленные индексы и значения. Иногда Visual Studio .NET позволяет определить переменные-члены в классе клиентского окна, которые сопоставляются со свойствами элементов ActiveX, имеющихся в клиентском классе. Сгенерированный DDX-код (Dialog Data Exchange) осуществляет обмен информацией между свойствами элемента ActiveX и переменными-членами клиентского класса.

Методы элементов управления ActiveX похожи на функции. У метода есть символьное имя, набор параметров и возвращаемое значение. Для вызова метода вызывается функция-член класса С++, представляющего данный элемент ActiveX. Разработчик элемента управления может определить любые нужные методы, например, *PreviousYear*, *LowerControlRods* и т. д.

В отличие от обычных элементов управления элемент ActiveX не посылает своему контейнеру уведомляющих сообщений с префиксом «WM_» — вместо этого он «инициирует события» (fire an event). У события есть символическое имя, и оно может содержать произвольный набор параметров — в сущности это функция контейнера, которую вызывает элемент управления ActiveX. Как и уведомляющие сообщения от обычных элементов управления события не возвращают в элемент ActiveX данных. В качестве примеров событий назовем *Click* (щелчок), *KeyDown* (нажатие клавиши) и *NewMonth* (новый месяц). События сопоставляются клиентскому классу так же, как и уведомляющие сообщения от элементов управления.

В мире MFC элементы управления ActiveX ведут себя как обычные дочерние окна, однако между окном контейнера и окном элемента управления есть весьма «толстая» прослойка кода. Кроме того, у элемента ActiveX окна может и не быть. При вызове *Create* само окно элемента управления не создается — вместо этого загружается его код, и ему выдается команда для активизации «на месте». После этого элемент ActiveX формирует свое окно, доступ к которому MFC обеспечивает через указатель *CWnd*. Однако настоятельно не рекомендуется применять в клиентской программе описатель *hWnd* элемента управления ActiveX.

Элементы ActiveX хранятся в отдельных DLL-библиотеках, причем такие DLL обычно имеют расширение .ocx. Приложение загружает DLL по мере надобности, используя изощренные приемы COM-технологии, основанные на операциях с реестром Windows. Но пока вам достаточно знать одно: элемент ActiveX, указанный на этапе разработки программы, загружается в период ее выполнения. Естественно, при поставках приложения, требующего определенных элементов управления ActiveX, вы должны включить в дистрибутивный комплект OCX-файлы и соответствующую программу установки.

Установка элементов управления ActiveX

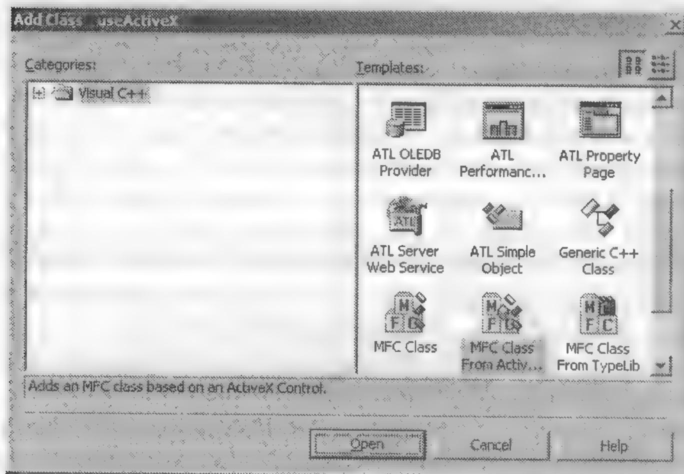
Допустим, вы раздобыли «навороченный» элемент ActiveX и хотите задействовать его в своем проекте. В первую очередь нужно скопировать соответствующую DLL на жесткий диск. Ее можно разместить где угодно, но элементы управления ActiveX

легче отслеживать, если все они размещены в одном месте, например, в системном каталоге (обычно \WINDOWS\SYSTEM в Microsoft Windows 95 и \WINNT\SYSTEM32 в Microsoft Windows 2000/XP). Скопируйте в тот же каталог файлы справки (HLP) и лицензирования (LIC).

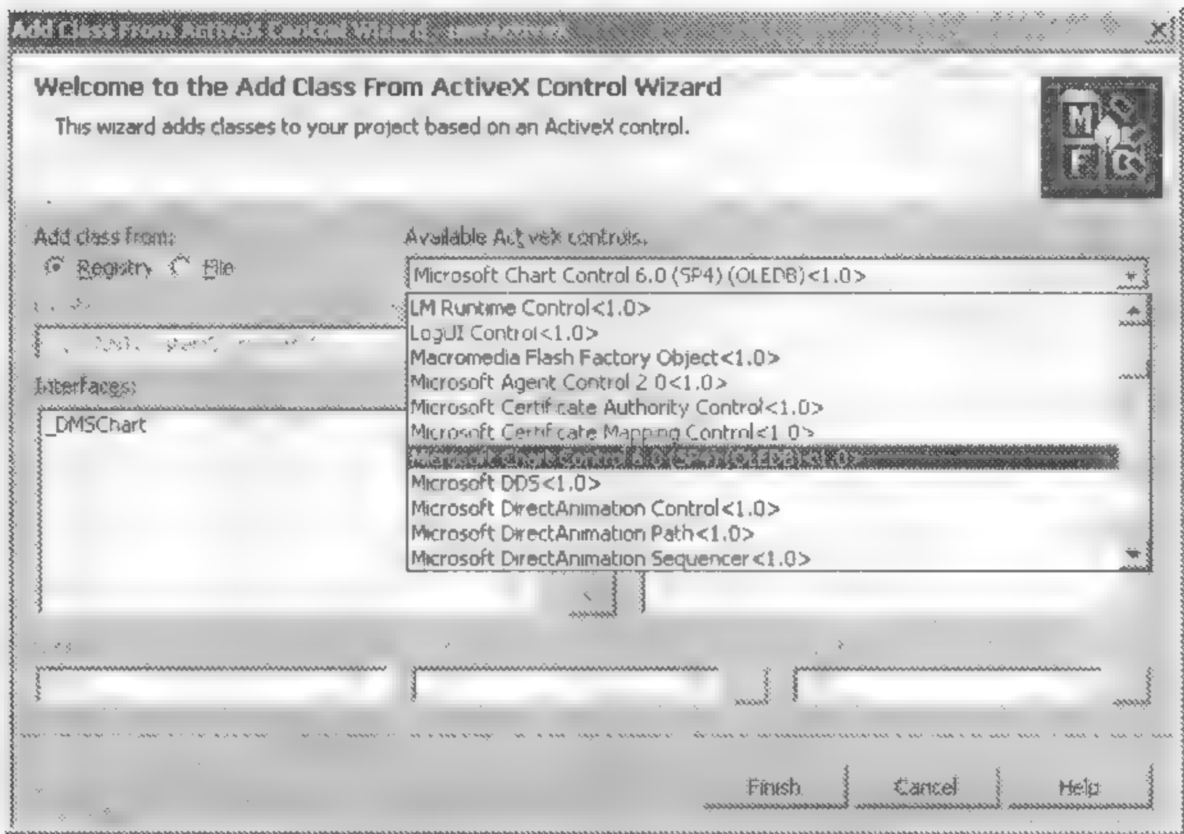
Следующий шаг — регистрация элемента управления в реестре Windows. На самом деле элемент ActiveX регистрирует себя сам, когда клиент вызывает специальную экспортируемую функцию. Обычно в качестве такого клиента выступает утилита Regsvr32, которая получает имя элемента управления из командной строки. Эта утилита годится для сценариев установки. В проекте REGCOMP на компакт-диске есть программа RegComp, позволяющая найти на диске заданный элемент управления. Некоторые элементы ActiveX требуют лицензирования, для чего иногда нужны дополнительные записи в реестре. (О работе с реестром Windows см. главы 15, 17, 24 и 25.) Лицензируемые элементы управления ActiveX обычно поставляются с программой установки, которая и заботится о создании соответствующих записей.

Зарегистрированный элемент ActiveX вы должны установить *в каждый проект*, в котором он используется. Это не означает, что нужно копировать OCX-файл — Visual Studio .NET создаст копию класса C++, оболочки соответствующего элемента управления, и последний появится в палитре элементов управления редактора диалоговых окон данного проекта.

Для установки элемента управления ActiveX выберите команду Add Class в меню Project, а в открывшемся диалоговом окне — MFC Class From ActiveX Control:



Выберите нужный элемент ActiveX в окне мастера Add Class From ActiveX Control Wizard. Вы увидите список всех элементов управления ActiveX, установленных в вашей системе в данный момент. Вот пример списка:



Элемент управления Calendar

Файл MSCal.ocx — популярный элемент ActiveX Microsoft Calendar, который скорее всего уже установлен и зарегистрирован на вашем компьютере. Если нет, не огорчайтесь — он есть на компакт-диске.

На рис. 9-1 показан элемент управления Calendar, расположенный внутри модального диалогового окна.

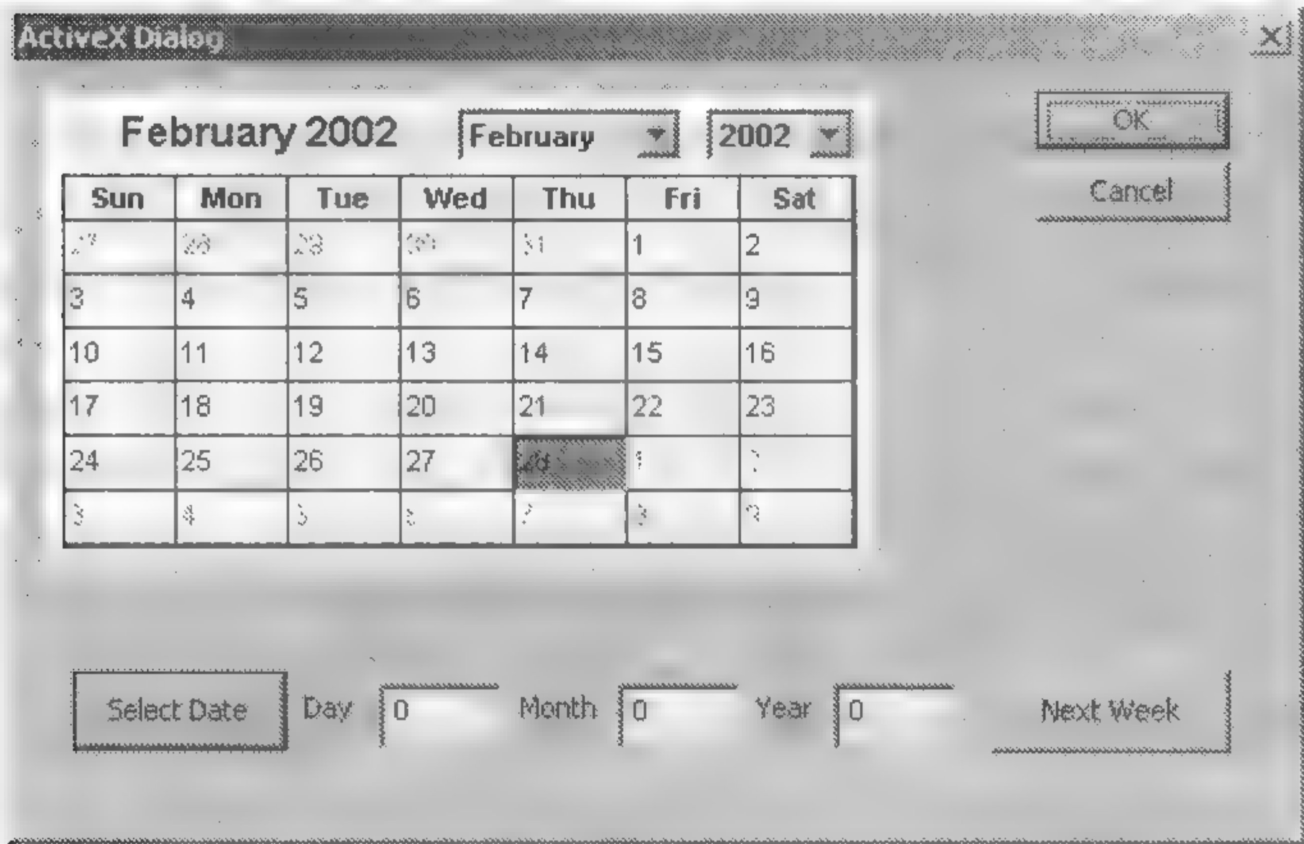


Рис. 9-1. *Использование элемента управления Calendar*

С этим элементом управления поставляется файл справки, в котором перечислены его свойства, методы и события (табл. 9-1).

Табл. 9-1. Свойства, методы и события элемента управления Calendar

Свойства	Методы	События
BackColor	*AboutBox*	*AfterUpdate*
Day	*NextDay*	*BeforeUpdate*
DayFont	*NextMonth*	*Click*
DayFontColor	*NextWeek*	*DblClick*

см. след. стр.

Табл. 9-1. (*продолжение*)

Свойства	Методы	События
DayLength	*NextYear*	*KeyDown*
FirstDay	*PreviousDay*	*KeyPress*
GridCellEffect	*PreviousMonth*	*KeyUp*
GridFont	*PreviousWeek*	*NewMonth*
GridFontColor	*PreviousYear*	*NewYear*
GridLinesColor	*Refresh*	
Month	*Today*	
MonthLength		
ShowDateSelectors		
ShowDays		
ShowHorizontalGridlines		
ShowTitle		
ShowVerticalGridlines		
TitleFont		
TitleFontColor		
Value		
ValueIsNull		
Year		

В примере Ex09a мы задействуем свойства *BackColor*, *Day*, *Month*, *Year* и *Value*. *BackColor* — переменная типа unsigned long, но используется как тип *OLE_COLOR*, а это почти то же, что и *COLORREF*. *Day*, *Month* и *Year* имеют тип short integer. *Value* — специальный тип *VARIANT* (он описан в главе 25). Дата хранится в нем в виде 64-разрядного значения.

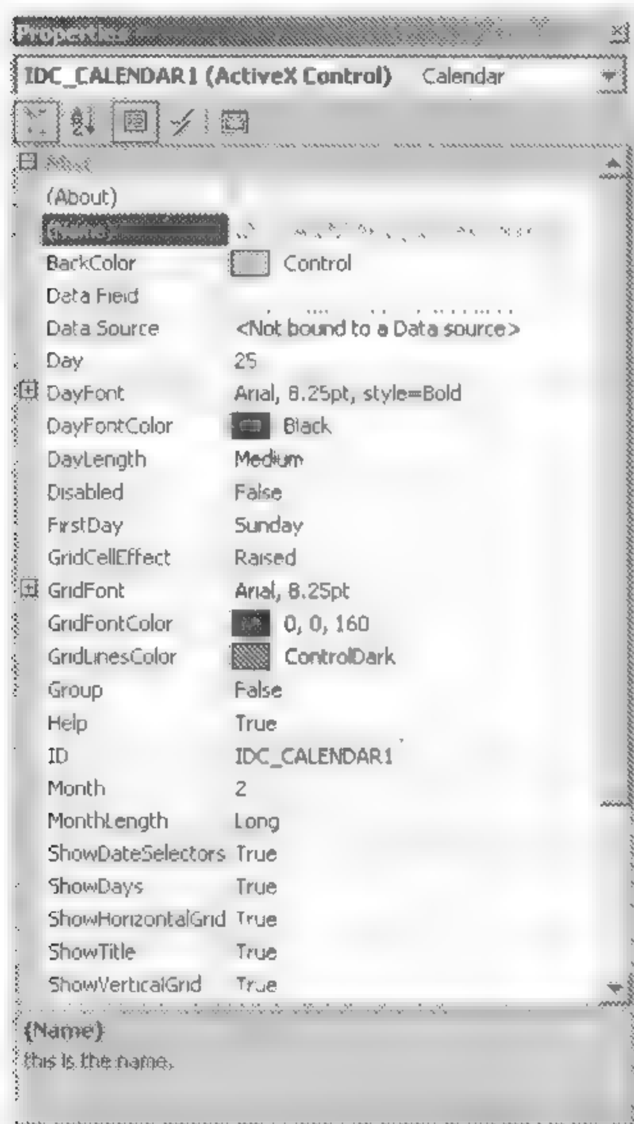
У каждого из перечисленных свойств, методов и событий есть соответствующий целочисленный идентификатор. Сведения об именах, типах, последовательности аргументов и идентификаторах хранятся в элементе управления, откуда их извлекает Visual Studio .NET во время конструирования контейнера.

Программирование контейнера ActiveX-элементов

MFC и Visual Studio .NET поддерживают применение элементов ActiveX как в диалоговых окнах, так и в виде «дочерних окон». Чтобы задействовать элемент управления ActiveX, надо знать, как он предоставляет доступ к своим свойствам и как ваш DDX-код взаимодействует со значениями этих свойств.

Доступ к свойствам

Набор свойств, доступных на этапе разработки, определяет создатель элемента ActiveX. Эти свойства отображаются на страницах свойств, которые элемент управления выводит в редакторе диалоговых окон, если щелкнуть его правой кнопкой и в контекстном меню выбрать Properties. Основная страница свойств элемента управления Calendar выглядит так:



В период выполнения доступны все свойства элемента ActiveX, в том числе и те, что были доступны на этапе разработки. Однако некоторые свойства могут быть определены «только для чтения».

Классы-оболочки C++, создаваемые Visual Studio .NET для ActiveX-элемента

При добавлении в проект ActiveX-элемента среда Visual Studio .NET — в соответствии с набором свойств и методов данного элемента управления — генерирует класс-оболочку C++, производный от *CWnd*. Сгенерированный класс содержит функции-члены для всех свойств и методов, а также конструкторы, которые можно задействовать для динамического создания экземпляров данного элемента управления. (Visual Studio .NET также генерирует классы-оболочки объектов, используемых элементом управления.) Вот несколько типичных функций-членов из файла CCalendar.h, которые Visual Studio .NET генерирует для элемента управления Calendar:

```
unsigned long get_BackColor()
{
    unsigned long result;
    InvokeHelper(DISPID_BACKCOLOR,
        DISPATCH_PROPERTYGET, VT_UI4, (void*)&result, NULL);
    return result;
}
void put_BackColor(unsigned long newValue)
{
    static BYTE parms[] = VTS_UI4 ;
    InvokeHelper(DISPID_BACKCOLOR, DISPATCH_PROPERTYPUT,
            VT_EMPTY, NULL, parms, newValue);
}
```

```
short get_Day()
{
    short result;
    InvokeHelper(0x11, DISPATCH_PROPERTYGET,
            VT_I2, (void*)&result, NULL);
    return result;
}
void put_Day(short newValue)
{
    static BYTE parms[] = VTS_I2 ;
    InvokeHelper(0x11, DISPATCH_PROPERTYPUT, VT_EMPTY,
            NULL, parms, newValue);
}
LPDISPATCH get_DayFont()
{
    LPDISPATCH result;
    InvokeHelper(0x1, DISPATCH_PROPERTYGET,
            VT_DISPATCH, (void*)&result, NULL);
    return result;
}
void put_DayFont(LPDISPATCH newValue)
{
    static BYTE parms[] = VTS_DISPATCH ;
    InvokeHelper(0x1, DISPATCH_PROPERTYPUT,
            VT_EMPTY, NULL, parms, newValue);
}
unsigned long get_DayFontColor()
{
    unsigned long result;
    InvokeHelper(0x2, DISPATCH_PROPERTYGET, VT_UI4,
            (void*)&result, NULL);
    return result;
}
void put_DayFontColor(unsigned long newValue)
{
    static BYTE parms[] = VTS_UI4 ;
    InvokeHelper(0x2, DISPATCH_PROPERTYPUT,
            VT_EMPTY, NULL, parms, newValue);
}
short get_DayLength()
{
    short result;
    InvokeHelper(0x12, DISPATCH_PROPERTYGET, VT_I2,
            (void*)&result, NULL);
    return result;
}
void put_DayLength(short newValue)
{
    static BYTE parms[] = VTS_I2 ;
```

```
   InvokeHelper(0x12, DISPATCH_PROPERTYPUT,
            VT_EMPTY, NULL, parms, newValue);
}
short get_FirstDay()
{
   short result;
   InvokeHelper(0x13, DISPATCH_PROPERTYGET,
            VT_I2, (void*)&result, NULL);
   return result;
}
void put_FirstDay(short newValue)
{
   static BYTE parms[] = VTS_I2 ;
   InvokeHelper(0x13, DISPATCH_PROPERTYPUT,
            VT_EMPTY, NULL, parms, newValue);
}

void NextDay()
{
   InvokeHelper(0x16, DISPATCH_METHOD,
            VT_EMPTY, NULL, NULL);
}
void NextMonth()
{
   InvokeHelper(0x17, DISPATCH_METHOD,
            VT_EMPTY, NULL, NULL);
}
void NextWeek()
{
   InvokeHelper(0x18, DISPATCH_METHOD,
            VT_EMPTY, NULL, NULL);
}
void NextYear()
{
   InvokeHelper(0x19, DISPATCH_METHOD,
            VT_EMPTY, NULL, NULL);
}
```

Насчет кода этих функций особо не беспокойтесь, однако заметьте, что первый параметр в вызовах каждой из *InvokeHelper*-функций можно сопоставить с диспетчерским идентификатором соответствующего свойства или метода в списке свойств элемента управления Calendar. Как видите, для каждого свойства существуют пары отдельных функций «*get_*» (получить) и «*set_*» (установить). Для вызова метода вы просто вызываете нужную функцию. Например, для вызова метода *NextDay* из функции класса диалогового окна можно написать так:

```
m_calendar.NextDay();
```

Здесь *m_calendar* — это объект класса *CCalendar*, служащего оболочкой для элемента управления Calendar.

Поддержка ActiveX-элементов в MFC Application Wizard

Если в MFC Application Wizard установлен флажок ActiveX Controls (а по умолчанию это так), в функцию-член *InitInstance* класса приложения вставляется строка:

```
AfxEnableControlContainer();
```

Кроме того, в файле StdAfx.h вашего проекта появляется строка:

```
#include <afxdisp.h>
```

Если вы решили использовать элементы ActiveX в существующем проекте, можете просто вставить эти две строки.

Мастер Add Class Wizard и диалоговое окно контейнера

Если вы создали шаблон диалогового окна с помощью редактора диалоговых окон, то знаете, что, используя Add Class Wizard, для этого окна можно сгенерировать класс C++. Если шаблон содержит ActiveX-элементы, Add Member Variable можно задействовать для добавления переменных-членов и обработчиков событий.

Переменные-члены класса диалогового окна или применение класса-оболочки

Какого типа переменные-члены добавлять в диалоговое окно для элемента управления ActiveX? Чтобы установить свойство элемента управления до вызова *DoModal* для этого диалогового окна, можно добавить переменную, соответствующую этому свойству. А чтобы менять свойства внутри функций-членов диалогового окна, придется добавить переменную-объект класса-оболочки элемента управления ActiveX.

Сейчас самое время вспомнить логику работы DDX MFC. Вернемся к диалоговому окну Ex06a из главы 8. Функция *CDialog::OnInitDialog* вызывает *CWnd::UpdateData(FALSE)* для считывания значений переменных-членов, а функция *CDialog::OnOK* вызывает *UpdateData (TRUE)* для записи в эти переменные. Допустим, вы добавили по одной переменной для каждого свойства элемента управления ActiveX и хотите получить значение свойства Value в обработчике сообщения от кнопки. Если вызвать *UpdateData(FALSE)*, считаются значения *всех* свойств *всех* элементов управления диалогового окна — ясно, что это излишняя трата времени. Эффективнее не использовать переменную для каждого свойства, а вместо этого вызвать функцию с префиксом «*get_*» класса-оболочки. Для этого надо, используя Visual Studio .NET, создать переменную-объект класса-оболочки.

Предположим, у нас есть класс-оболочка *CCalendar* для элемента управления Calendar, а в классе диалогового окна — переменная-член *m_calendar*. Тогда значение свойства Value можно получить так:

```
COleVariant var = m_calendar.GetValue();
```

Примечание Тип *VARIANT* и класс *COleVariant* описаны в главе 23.

Теперь рассмотрим другой случай: вы хотите перед отображением элемента управления установить 5-й день месяца. Для этого добавьте в класс диалогового окна переменную-член *m_sCalDay* типа short integer, соответствующую свойству *Day*. Затем добавьте в функцию *DoDataExchange* вызов:

```
DDX_OCShort(pDX, ID_CALENDAR1, 0x11, m_sCalDay);
```

Третий параметр — идентификатор свойства *Day*, который можно найти в функциях *get_Day* и *put_Day*, созданных средой Visual Studio .NET для элемента управления. Диалоговое окно создается и отображается так:

```
CMyDialog dlg;
dlg.m_sCalDay = 5;
dlg.DoModal();
```

Перед отображением элемента управления на экране DDX-код позаботится об установке значения свойства в соответствии со значением переменной-члена. Дополнительного программирования это не требует. DDX-код при щелчке кнопки OK, конечно, установит значение переменной-члена в соответствии со значением свойства.

Примечание Даже если Visual Studio .NET правильно определит свойства элемента управления, она не обязательно создаст переменные-члены для всех свойств. В частности, не существует DDX-функций для свойств типа *VARIANT*, подобных свойству *Value* элемента Calendar. Для таких свойств понадобится класс-оболочка.

Привязка событий элементов ActiveX

Окно Properties, связанное с окном Class View позволяет сопоставлять события от элементов ActiveX с функциями-членами так же, как это делается для Windows-сообщений и командных сообщений от элементов управления. Если в классе диалогового окна есть элементы ActiveX, мастера, доступные из окна Properties, создают и поддерживают *карту приема событий* (event sink map), в которой определено соответствие между событиями и их функциями-обработчиками. С конкретным кодом в файлах ActiveXDialog.h и ActiveXDialog.cpp мы познакомимся попозже.

Примечание У элементов ActiveX есть неприятная особенность — инициировать события до того, как программа готова их получить. Если в обработчике события используются окна или указатели на объекты C++, то прежде, чем обращаться к ним, надо проверить, существуют ли эти объекты.

Закрепление элементов ActiveX в памяти

Обычно элемент ActiveX проецируется на адресное пространство вашего процесса только на время, пока активно его родительское диалоговое окно. Это значит, что всякий раз, когда пользователь открывает это окно, элемент приходится загружать заново. Благодаря дисковому кэшу повторная загрузка проходит быстрее первоначальной. Тем не менее, чтобы повысить производительность, можно закрепить (оставить) элемент ActiveX в памяти. Для этого в переопределенную функцию *OnInitDialog* после вызова функции базового класса добавьте строку:

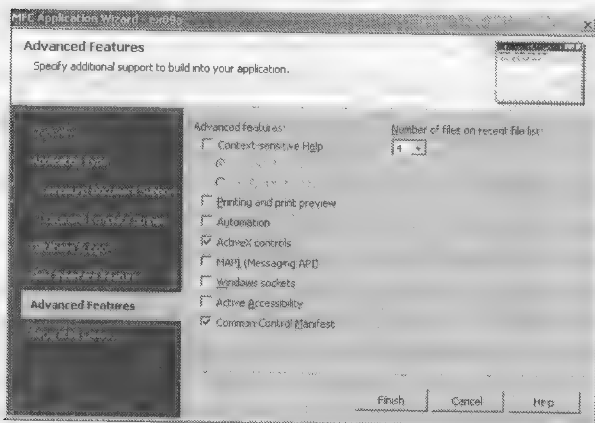
```
AfxOleLockControl(m_calendar.GetClsid());
```

Элемент ActiveX останется спроецированным на адресное пространство процесса до завершения программы или вызова функции *AfxOleUnlockControl*.

Пример Ex09a: контейнер ActiveX

Пора создать приложение, использующее элемент управления Calendar в диалоговом окне.

1. **Зарегистрируйте элемент управления Calendar.** Если элемента управления нет в системной папке, скопируйте файлы MSCal.ocx, MSCal.hlp и MSCal.cnt в системный каталог и зарегистрируйте элемент управления, запустив REGCOMP.
2. **С помощью MFC Application Wizard создайте проект Ex09a.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения. Проверьте, установлен ли флажок ActiveX controls:



3. **Установите элемент управления Calendar в проект Ex09a.** В меню Project выберите команду Add Class. В окне Add Class выберите MFC Class From ActiveX и щелкните Open. В окне Add Class From ActiveX Control Wizard в списке доступных элементов управления выберите Calendar Control 9.0, как показано ниже на рисунке Visual Studio .NET создаст соответствующий класс в каталоге Ex09a.
4. **Отредактируйте класс элемента управления Calendar для обработки справочных сообщений.** Добавьте в Calendar.cpp код таблицы сообщений:

```
BEGIN_MESSAGE_MAP(CCalendar, CWnd)
   ON_WM_HELPINFO()
END_MESSAGE_MAP()
```

   В тот же файл добавьте функцию *OnHelpInfo*:

```
BOOL CCalendar::OnHelpInfo(HELPINFO* pHelpInfo)
{
// Отредактируйте эту строку в соответствии с особенностями вашей системы
::WinHelp(GetSafeHwnd(), "c:\\winnt\\system32\\mscal.hlp",
```

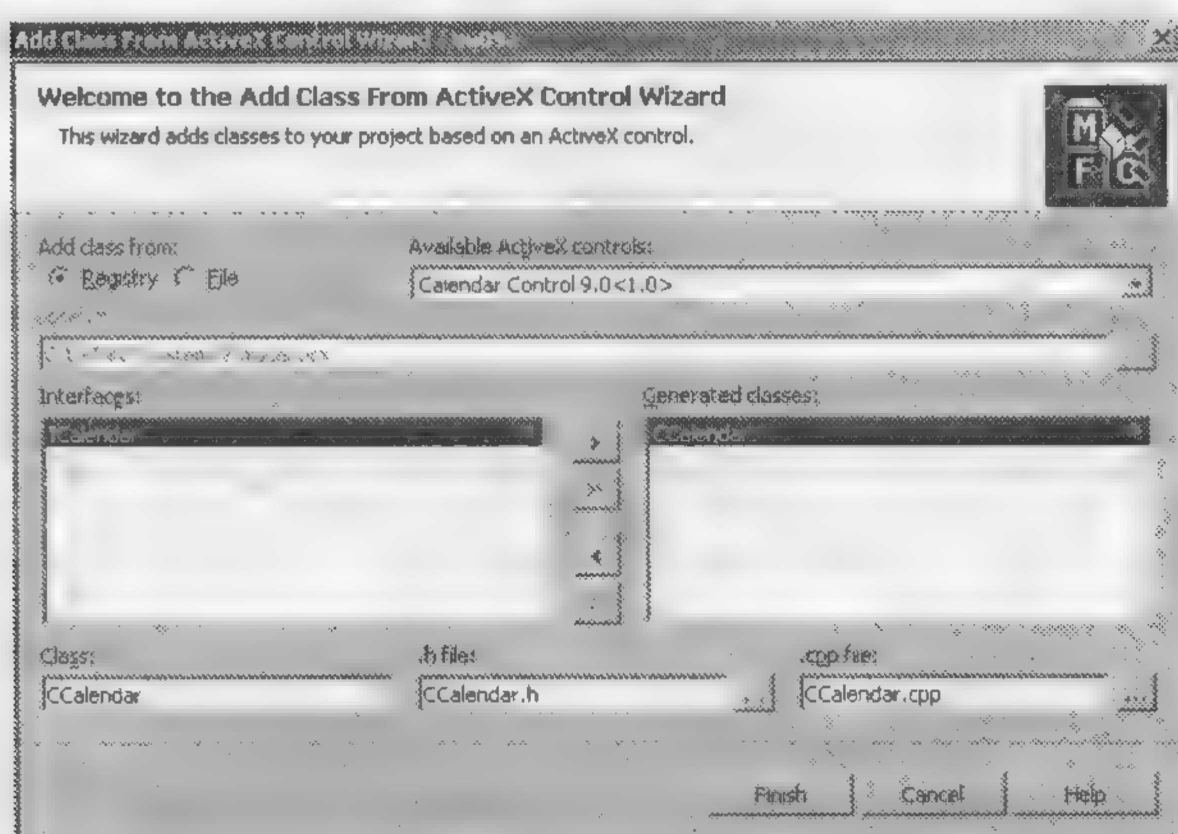
```
HELP_FINDER, 0);
return FALSE;
}
```

В Calendar.h добавьте прототип функции и объявление таблицы сообщений:

```
protected:
    afx_msg BOOL OnHelpInfo(HELPINFO* pHelpInfo);
    DECLARE_MESSAGE_MAP()
```

Функция *OnHelpInfo* вызывается, когда пользователь нажимает клавишу F1, при условии что фокус ввода установлен на элементе управления Calendar. Мы вынуждены добавлять код вручную, так как Visual Studio .NET не изменяет сгенерированные классы элементов ActiveX.

Примечание Макрос *ON_WM_HELPINFO* служит для привязки сообщения *WM_HELP*. Вы можете применять *ON_WM_HELPINFO* в любом классе диалогового окна или окна представления, чтобы затем вызывать в его обработчике любую справочную систему.



5. **С помощью редактора диалоговых окон создайте новый ресурс диалогового окна.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс. Измените идентификатор по умолчанию на *IDD_ACTIVEXDIALOG*, свойству Caption присвойте значение ActiveX Dialog, а свойству Context Help — TRUE. Оставьте созданные по умолчанию кнопки OK и Cancel с идентификаторами *IDOK* и *IDCANCEL* и добавьте другие элементы управления в соответствии с рис. 9-1. Сделайте кнопку Select Date кнопкой по умолчанию. Щелкните диалоговое окно правой кнопкой и в контекстном меню выберите Insert ActiveX Control, а в списке — элемент управления Calendar. Задайте порядок обхода по нажатию клавиши Tab. Присвойте элементам управления идентификаторы в соответствии с таблицей:

Элемент управления	Идентификатор
Calendar	*IDC_CALENDAR1*
Кнопка Select Date	*IDC_SELECTDATE*
Поле ввода	*IDC_DAY*
Поле ввода	*IDC_MONTH*
Поле ввода	*IDC_YEAR*
Кнопка Next Week	*IDC_NEXTWEEK*

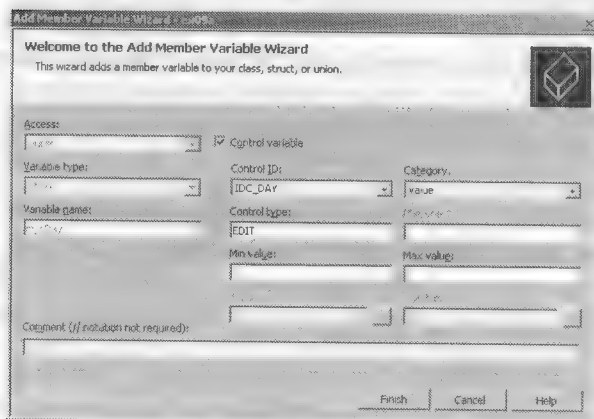
6. **Используя Visual Studio .NET, создайте класс *CActiveXDialog.*** В меню Project выберите команду Add Class. В окне MFC Class Wizard создайте класс *CActiveXDialog,* производный от *CDialog* и основанный на шаблоне *IDD_ACTIVEXDIALOG.* Обязательно в качестве базового выберите класс *CDialog*!

   Щелкните правой кнопкой класс *CActiveXDialog* в Class View и в контекстном меню выберите Properties. В окне Properties щелкните кнопку Overrides и создайте переопределенные функции *OnInitDialog* и *OnOK.*

   В окне Properties щелкните кнопку Events и добавьте перечисленные в таблице функции-обработчики. Чтобы добавить обработчик события, в открывшемся списке выберите нужную, щелкните стрелку «вниз» рядом с ней и выберите команду с приставкой <Add>. Функция откроется в редакторе кода. Повторите эту последовательность операций для каждого обработчика.

Идентификатор объекта	Сообщение	Функция-член
IDC_CALENDAR1	*NewMonth*	*NewMonthCalendar1*
IDC_SELECTDATE	*BN_CLICKED*	*OnBnClickedSelectdate*
IDC_NEXTWEEK	*BN_CLICKED*	*OnBnClickedNextweek*

7. **Используя Add Member Variable Wizard, добавьте в класс *CActiveXDialog* переменные-члены.** Щелкните правой кнопкой класс CActiveXDialog в окне Class View, в контекстном меню выберите Add Variable и добавьте переменные-члены *m_calendar, m_sDay, m_sMonth* и *m_sYear.*



Примечание Вы могли бы подумать, что вкладка ActiveX Events предназначена для привязки событий элементов ActiveX к функциями контейнера. Но это не так, разработчики используют ее для *определения* событий создаваемого элемента управления ActiveX.

8 **Отредактируйте класс *CActiveXDialog*.** Добавьте переменные-члены *m_varValue* и *m_BackColor* и затем отредактируйте код двух переопределенных функций и функций-обработчиков (*OnInitDialog*, *NewMonthCalendar1*, *OnBnClickedSelectdate*, *OnBnClickedNextweek* и *OnOK*). Далее показан весь код для класса диалогового окна — новый код выделен.

ActiveXDialog.H

```
#pragma once
#include "calendar.h"
////////////////////////////////////////////////////////////////////////
// CActiveXDialog dialog
class CActiveXDialog : public CDialog

{
    DECLARE_DYNAMIC(CActiveXDialog)
public:
    CActiveXDialog(CWnd* pParent = NULL);   // standard constructor
    virtual ~CActiveXDialog();

// Dialog Data
    enum { IDD = IDD_ACTIVEXDIALOG };

    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    DECLARE_MESSAGE_MAP()
public:

    virtual BOOL OnInitDialog();
    void NewMonthCalendar1();
    afx_msg void OnBnClickedSelectdate();
    afx_msg void OnBnClickedNextweek();

    DECLARE_EVENTSINK_MAP()
public:
    CCalendar m_calendar;
    short  m_sDay;
    short  m_sMonth;
    short  m_sYear;
    COleVariant m_varValue;
    unsigned long m_BackColor;

protected:
    virtual void OnOK();
};
```

ActiveXDialog.cpp

```
[illegible]
{
   [illegible]
   m_BackColor = 0x8000000F;
}

[illegible]
{
}

[illegible]
{
   [illegible]

   DDX_OCColor(pDX, IDC_CALENDAR1, DISPID_BACKCOLOR, m_BackColor);
}

[illegible]
}
```

```
    [illegible]
    m_calendar.put_Value(m_varValue); // DDX в типе VARIANT не поддерживается
    [illegible]
                  [illegible]
                  [illegible]
                  [illegible]

}
[illegible]
{
    [illegible]
    m_varValue = m_calendar.get_Value();    // DDX в типе VARIANT
                                            // не поддерживается
}

[illegible]
{
    AfxMessageBox("EVENT:  CActiveXDialog::NewMonthCalendar1");
}

[illegible]
{
    CDataExchange dx(this, TRUE);
    DDX_Text(&dx, IDC_DAY, m_sDay);
    DDX_Text(&dx, IDC_MONTH, m_sMonth);
    DDX_Text(&dx, IDC_YEAR, m_sYear);
    m_calendar.put_Day(m_sDay);
    m_calendar.put_Month(m_sMonth);
    m_calendar.put_Year(m_sYear);
}

[illegible]
{
    m_calendar.NextWeek();
}
```

Функция *OnBnClickedSelectdate* вызывается по щелчку кнопки Select Date, получает значения дня, месяца и года из полей ввода и передает их свойствам элемента управления. Add Member Variable Wizard не умеет создавать DDX-код для свойства *BackColor*, поэтому вам придется сделать это вручную. Кроме того, для типа *VARIANT* нет функций DDX, поэтому необходимо добавить в функции *OnInitDialog* и *OnOK* код, присваивающий и получающий дату в свойстве *Value*.

9. **Свяжите диалоговое окно с окном представления.** В окне Properties утилиты Class View создайте обработчик сообщения *WM_LBUTTONDOWN* и отредактируйте его:

```
void CEx09aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CActiveXDialog dlg;
    dlg.m_BackColor = RGB(255, 251, 240); // light yellow
```

```
    COleDateTime today = COleDateTime::GetCurrentTime();
    dlg.m_varValue = COleDateTime(today.get_Year(),
                                  today.get_Month(),
                                  today.get_Day(), 0, 0, 0);
    if (dlg.DoModal() == IDOK) {
        COleDateTime date(dlg.m_varValue);
        AfxMessageBox(date.Format("%B %d, %Y"));
    }
}
```

Код устанавливает светло-желтый цвет фона и текущий день в качестве используемой даты, показывает диалоговое окно и сообщает, какую дату вернул элемент управления Calendar. Вам нужно включить ActiveXDialog.h в Ex09aView.cpp.

10. **Измените виртуальную функцию *OnDraw* в файле Ex09aView.cpp.** Чтобы предложить пользователю нажать левую кнопку мыши, добавьте в функцию *OnDraw* строку:

```
pDC->TextOut(0, 0, "Press the left mouse button here.");
```

11. **Соберите оттестируйте приложение Ex09a.** Откройте диалоговое окно, введите дату в три поля ввода и щелкните кнопку Select Date. Щелкните кнопку Next Week. Попытайтесь перенести выбранную дату прямо в новый месяц — вы увидите окно сообщения, вызываемое событием *NewMonth*. Окончательную дату вы увидите в другом окне сообщения, щелкнув кнопку OK.

Для тех, кто программирует в Win32

Просмотрев файл Ex09a.rc в текстовом редакторе, вы наверняка удивитесь. Вот запись для элемента управления Calendar из шаблона диалогового окна:

```
CONTROL      "",IDC_CALENDAR1,
             "{8E27C92B-1264-101C-8A2F-040224009C02}",
             WS_TABSTOP,7,7,217,113
```

Там, где должно быть имя оконного класса, указана последовательность из 32 цифр. В чем здесь дело? В том, что этот шаблон не передается прямо Windows. Функция *CDialog::DoModal* сначала «обрабатывает» шаблон ресурса, прежде чем передать его Windows-функции создания диалогового окна. Эта функция «вырезает» все элементы ActiveX, и диалоговое окно создается без них. Затем она загружает элементы ActiveX (по их числовым идентификаторам из 32 цифр, называемым CLSID) и активизирует их, в результате чего они создают свои окна в нужных местах. Начальные значения свойств, которые задаются в графическом редакторе, хранятся в исполняемом файле проекта в двоичной форме как нестандартный ресурс DLGINIT.

В процессе функционирования модального диалогового окна MFC-код обрабатывает сообщения, посылаемые ему как обычными элементами управления, так и элементами ActiveX. Это позволяет переключаться между всеми элементами управления диалогового окна клавишей Tab, хотя элементы ActiveX и не входят в фактически используемый шаблон диалогового окна.

Вызывая функцию-член объекта ActiveX, вы можете решить, что обращаетесь к функции дочернего окна. Но окно элемента управления ActiveX уже давно удалено из шаблона диалогового окна — это MFC создает иллюзию работы с настоящим дочерним окном. В терминологии ActiveX у контейнера есть *посредник* (site), который окном не является. Вы вызываете функции посредника, а ActiveX и MFC обеспечивают связь с окном в элементе ActiveX.

Окно контейнера — это объект класса, производного от *CWnd*. Посредник элемента управления — также объект класса, производного от *CWnd*, класса-оболочки элемента управления ActiveX. Это значит, что в класс *CWnd* встроена поддержка и для контейнеров, и для посредников.

ActiveX-элементы в HTML-файлах

Вы видели ActiveX-элемент Calendar в модальном диалоговом окне MFC. Его можно использовать и на Web-странице. Следующий HTML-код будет работать, если на машине установлен и зарегистрирован элемент управления Calendar:

```
<OBJECT
   CLASSID="clsid:8E27C92B-1264-101C-8A2F-040224009C02"
   WIDTH=300 HEIGHT=200 BORDER=1 HSPACE=5 ID=calendar>
<PARAM NAME="Day" VALUE=7>
<PARAM NAME="Month" VALUE=11>
<PARAM NAME="Year" VALUE=1998>
</OBJECT>
```

Атрибут CLASSID, значение которого совпадает со значением в ресурсе диалогового окна из примера Ex09a, идентифицирует в реестре элемент управления Calendar, благодаря чему браузер может загрузить элемент ActiveX.

Создание элементов ActiveX в период выполнения

Вы видели, как с помощью редактора диалоговых окон добавлять элементы ActiveX в период разработки программы. Если же нужно создать элемент управления в период выполнения, не применяя шаблоны ресурсов, сделайте следующее.

1. Добавьте компонент в проект. Visual Studio .NET создаст файлы для класса-оболочки.
2. Добавьте в класс диалогового окна или другой оконный класс C++ переменную-член типа класс-оболочка. Внедренный таким образом объект C++ будет создаваться и разрушаться вместе с оконным объектом.
3. В меню View выберите Resource View. В окне Resource View щелкните правой кнопкой RC-файл и в контекстном меню выберите Resource Symbols. Добавьте константу-идентификатор для нового элемента управления.
4. Если родительское окно — диалоговое, то в окне Properties утилиты Class View переопределите функцию *CDialog::OnInitDialog*. Для других окон, кроме основанных на классе *CDialog*, сопоставьте сообщение *WM_CREATE*. В любом случае новая функция должна вызывать функцию-член *Create* внедренного эле-

мента управления ActiveX. Этот вызов неявно приведет к отображению нового элемента управления в диалоговом окне. Он уничтожается при уничтожении родительского окна.

5. В классе родительского окна вручную добавьте обработчики событий для нового элемента управления. Не забудьте добавить макросы таблицы приема событий.

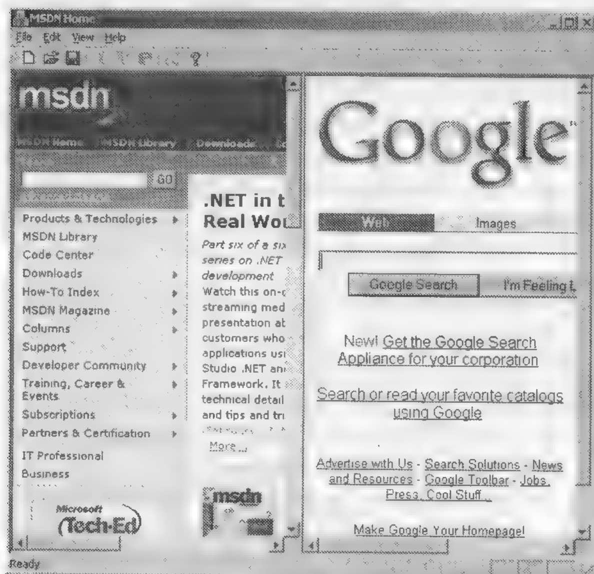
Совет Доступные в окне Properties мастера не помогут в работе с таблицей приема событий, если ActiveX-элемент создается в проекте динамически. Попробуйте вставить элемент управления в диалог в другом, временном проекте. Настроив таблицу приема всех событий, скопируйте код в класс родительского окна в основном проекте.

Пример Ex09b: ActiveX-элемент в браузере

Основная часть функциональности браузера содержится в одном большом элементе ActiveX — Shdocvw.dll. Запуская Internet Explorer, вы вызываете маленькую программу-оболочку, которая загружает этот элемент управления Web-браузера в свое основное окно.

Примечание Полную документацию по свойствам, методам и событиям элемента управления WebBrowser вы найдете в интерактивной библиотеке MSDN в составе Visual Studio. NET.

Такая модульная архитектура позволяет вам очень легко написать свою программу-браузер. Ex09b создает двухоконный браузер, в котором выводится страница системы поиска, а рядом — обычная страница:



Окно представления содержит два элемента управления WebBrowser, которые занимают всю клиентскую область. Когда пользователь щелкает ссылку в поисковом (левом) элементе управления, программа обрабатывает эту команду и перенаправляет результат в целевой (правый) элемент управления.

1. **Убедитесь, что элемент управления WebBrowser зарегистрирован.** У вас, конечно, установлен Microsoft Internet Explorer самой последней версии, так как она необходима Visual Studio .NET, поэтому элемент управления WebBrowser должен быть зарегистрирован. Если нужно, загрузите Internet Explorer с сайта *http:\\www.microsoft.com*.
2. **С помощью MFC Application Wizard создайте проект Ex09b.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения. Проверьте, установлен ли флажок ActiveX Controls, как в примере Ex09a.
3. **Установите элемент управления WebBrowser в проект Ex09b.** Выберите команду Add Class в меню Project и в открывшемся диалоговом окне выберите MFC Class From ActiveX Control. В открывшемся списке выберите элемент Microsoft Web Browser. Visual Studio .NET предложит создать классы-оболочки для двух интерфейсов: *IWebBrowser* и *IWebBrowser2*. Выберите *IWebBrowser2*. Visual Studio .NET создаст класс-оболочку для *IWebBrowser2* и добавит соответствующие файлы в проект.
4. **Добавьте в класс *CEx09bView* две переменные-члены типа *CWebBrowser*.** Проще всего это сделать вручную, добавив в заголовочный файл строки:

```
private:
    CWebBrowser2 m_target;
    CWebBrowser2 m_search;
```

   Не забудьте проверить наличие оператора *#include* для файла CWebBrowser2.h.
5. **Добавьте константы-идентификаторы дочерних окон для обоих элементов управления.** Щелкните правой кнопкой класс «вид» в Resource View и в контекстном меню выберите Resource Symbols. Добавьте символы *ID_BROWSER_SEARCH* и *ID_BROWSER_TARGET*.
6. **Добавьте статическую переменную-член типа массив символов для хранения URL-адреса поисковой системы Google.** В объявление класса в Ex09bView.h добавьте статическую переменную:

```
private:
    static const char s_engineGoogle[];
```

   Затем в файл Ex09bView.cpp вне кода всех функций добавьте определение:

```
const char CEx09bView::s_engineGoogle[] = "http://www.google.com/";
```

7. **Используя окно Properties утилиты Class View, создайте обработчики сообщений *WM_CREATE* и *WM_SIZE* для класса «вид».** Затем измените код обработчиков в Ex09bView.cpp:

```
int CEx09bView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;

   DWORD dwStyle = WS_VISIBLE | WS_CHILD;
   if (m_search.Create(NULL, dwStyle, CRect(0, 0, 100, 100),
                  this, ID_BROWSER_SEARCH) == 0) {
      AfxMessageBox("Unable to create search control!\n");
      return -1;
   }
   m_search.Navigate(s_engineGoogle, NULL, NULL, NULL, NULL);

   if (m_target.Create(NULL, dwStyle, CRect(0, 0, 100, 100),
                  this, ID_BROWSER_TARGET) == 0) {
      AfxMessageBox("Unable to create target control!\n");
      return -1;
   }
   m_target.GoHome(); // как задано в параметрах Internet Explorer

   return 0;
}

void CEx09bView::OnSize(UINT nType, int cx, int cy)
{
   CView::OnSize(nType, cx, cy);

   CRect rectClient;
   GetClientRect(rectClient);
   CRect rectBrowse(rectClient);
   rectBrowse.right = rectClient.right / 2;
   CRect rectSearch(rectClient);
   rectSearch.left = rectClient.right / 2;

   m_target.put_Width(rectBrowse.right - rectBrowse.left);
   m_target.put_Height(rectBrowse.bottom - rectBrowse.top);
   m_target.UpdateWindow();

   m_search.put_Left(rectSearch.left);
   m_search.put_Width(rectSearch.right - rectSearch.left);
   m_search.put_Height(rectSearch.bottom - rectSearch.top);
   m_search.UpdateWindow();
}
```

Функция *OnCreate* создает внутри окна представления два окна браузера. Правое отображает основную страницу системы поиска Google, а левое — домашнюю страницу, заданную в параметрах Интернета в панели управления. Функция *OnSize*, вызываемая при всяком изменении размера окна представления, гарантирует, что окна браузера полностью заполняют окно класса «вид». Функции-члены *put_Width* и *put_Height* класса *CWebBrowser2* позволяют установить свойства Width (ширина) и Height (высота).

8. **Добавьте в файлы Ex09bView макросы приема событий.** Мастера, доступные в окне Properties, не могут выполнить привязку событий для динамически создаваемых элементов ActiveX, поэтому придется сделать это вручную. Добавьте внутрь объявления класса в файле Ex09bView.h строки:

```
protected:
    afx_msg void OnBeforeNavigateExplorer1(LPCTSTR URL,
        long Flags, LPCTSTR TargetFrameName,
        VARIANT FAR* PostData, LPCTSTR Headers, BOOL FAR* Cancel);
    afx_msg void OnTitleChangeExplorer2(LPCTSTR Text);
    DECLARE_EVENTSINK_MAP()
```

Затем добавьте этот код в файл Ex09bView.cpp:

```
BEGIN_EVENTSINK_MAP(CEx09bView, CView)
    ON_EVENT(CEx09bView, ID_BROWSER_SEARCH, 100,
        OnBeforeNavigateExplorer1, VTS_BSTR VTS_I4 VTS_BSTR
        VTS_PVARIANT VTS_BSTR VTS_PBOOL)
    ON_EVENT(CEx09bView, ID_BROWSER_TARGET, 113,
        OnTitleChangeExplorer2, VTS_BSTR)
END_EVENTSINK_MAP()
```

9. **Добавьте две функции-обработчики событий.** Добавьте в файл Ex09b-View.cpp функции-члены:

```
void CEx09bView::OnBeforeNavigateExplorer1(LPCTSTR URL,
    long Flags, LPCTSTR TargetFrameName,
    VARIANT FAR* PostData, LPCTSTR Headers, BOOL FAR* Cancel)
{
    TRACE("CEx09bView::OnBeforeNavigateExplorer1 -URL = %s\n", URL);

     if (!strnicmp(URL, s_engineGoogle, strlen(s_engineGoogle))) {
        return;
    }
    m_target.Navigate(URL, NULL, NULL, PostData, NULL);
    *Cancel = TRUE;
}

void CEx09bView::OnTitleChangeExplorer2(LPCTSTR Text)
{
    // Осторожно! Событие может быть получено раньше,
    // чем мы будем готовы.
    CWnd* pWnd = AfxGetApp()->m_pMainWnd;
    if (pWnd != NULL) {
        if (::IsWindow(pWnd->m_hWnd)) {
            pWnd->SetWindowText(Text);
        }
    }
}
```

Обработчик *OnBeforeNavigateExplorer1* вызывается, когда пользователь щелкает ссылку на поисковой странице. Функция сравнивает URL-адрес ссылки (в строковом параметре *URL*) с URL-адресом поисковой системы. Если они со-

впадают, навигация продолжается в окне поиска, в противном случае она прекращается, и вызывается метод Navigate для целевого окна. Обработчик *OnTitleChangeExplorer2* обновляет заголовок окна Ex09b в соответствии с заголовком целевой страницы.

10. **Соберите и протестируйте приложение Ex09b.** Выполните какой-нибудь поиск на странице Google, затем просмотрите информацию, появившуюся на целевой странице.

Свойства-картинки

Некоторые элементы управления ActiveX поддерживают *свойства-картинки* (picture property), значениями которых могут быть растровые изображения, метафайлы и значки. Если у ActiveX-элемента есть хоть одно такое свойство, при установке данного элемента управления в проекте Visual Studio .NET создаст класс *CPicture*. Вы не обязаны использовать его, но вынуждены применять MFC-класс *CPictureHolder*. Для доступа к объявлению и коду класса *CPictureHolder* добавьте в StdAfx.h строку:

```
#include <afxctl.h>
```

Допустим, у вас есть ActiveX-элемент со свойством-картинкой по имени Picture. Вот как инициализировать это свойство растровым изображением из ресурса:

```
CPictureHolder pict;
pict.CreateFromBitmap(IDB_MYBITMAP); // из ресурса проекта
m_control.SetPicture(pict.GetPictureDispatch());
```

Примечание Подключив файл AfxCtl.h, вы не сможете статически скомпоновать свою программу с MFC-библиотекой. Чтобы создать автономную программу, поддерживающую свойства-картинки, вам придется позаимствовать код класса *CPictureHolder*, содержащийся в файле the \Program Files\Microsoft Visual Studio .NET\VC7\atlmfc\src\mfc\ctlpict.cpp.

Связываемые свойства: уведомления об изменении

Если ActiveX-элемент имеет свойство, обозначенное как *связываемое* (bindable), он посылает своему контейнеру уведомление *OnChanged* при каждом изменении значения свойства внутри элемента управления. Кроме того, он может послать уведомление *OnRequestEdit* для свойства, чье значение должно измениться, но пока этого не произошло[1]. Если контейнер вернет FALSE из обработчика *OnRequestEdit*, элемент управления должен отказаться от изменения значения свойства.

MFC полностью поддерживает уведомления об изменении свойств для контейнеров элементов ActiveX, однако в версии Visual C++ .NET соответствующая под-

[1] Для этого свойство следует обозначить как «запрашивающее разрешение на обновление» (requestedit). — *Прим. перев.*

держка со стороны мастеров отсутствует. Это значит, что вам придется вручную добавлять записи в таблицу приема событий в классе контейнера.

Допустим, у вас есть элемент управления ActiveX со связываемым свойством Note, идентификатор которого равен 4. Тогда вы должны добавить в таблицу приема событий макрос *ON_PROPNOTIFY*:

```
BEGIN_EVENTSINK_MAP(CAboutDlg, CDialog)
   ON_PROPNOTIFY(CAboutDlg, IDC_MYCTRL1, 4, OnNoteRequestEdit,
         OnNoteChanged)
END_EVENTSINK_MAP()
```

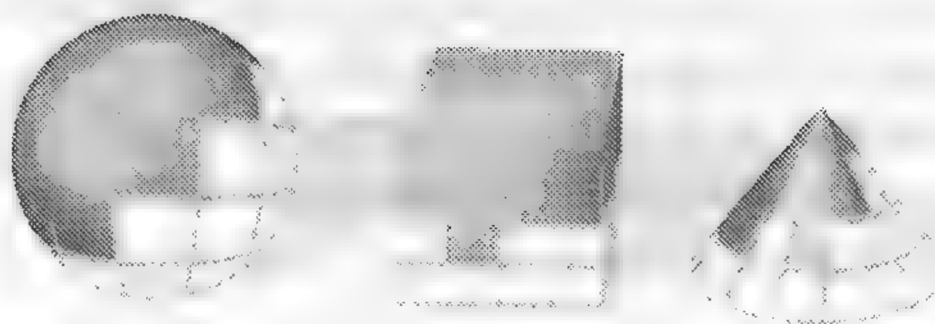
Затем вы должны написать код функций *OnNoteRequestEdit* и *OnNoteChanged*, возвращаемые значения и параметры которых должны точно соответствовать следующим:

```
BOOL CMyDlg::OnNoteRequestEdit(BOOL* pb)
{
   TRACE("CMyDlg::OnNoteRequestEdit\n");
   *pb = TRUE; // TRUE означает разрешение на изменение значения
   return TRUE;
}

BOOL CMyDlg::OnNoteChanged()
{
   TRACE("CMyDlg::OnNoteChanged\n");
   return TRUE;
}
```

Соответствующие прототипы надо добавить в заголовочный файл класса:

```
afx_msg BOOL OnNoteReqestEdit(BOOL* pb);
afx_msg BOOL OnNoteChanged();
```



ГЛАВА

10

Управление памятью в Win32

За истекшие годы Microsoft Windows претерпела массу перемен. В конце 1980-х системная память стоила бешеных денег, и приходилось выжимать максимум возможного из каждого байта оперативной памяти. Переход на 32 разряда коренным образом изменил ситуацию. В Win16 надо было выполнять массу дополнительных операций при вызове функций управления памятью (вроде *GlobalAlloc* и *GlobalLock*). Эти функции перенесены в Win32, но исключительно из соображений обратной совместимости. По сути же эти функции устроены совершенно иначе, да и появилось множество новых функций.

Мы начнем эту главу с хорошей порции теории, в том числе расскажем о фундаментальных функциях управления динамически распределяемой памятью, или *кучей* (heap). Затем вы увидите, как операторы new и delete языка C++ связываются с «нижележащими» функциями управления кучей. Наконец, вы узнаете, как использовать функции для *спроецированных в память файлов* (memory-mapped files), а также получите несколько советов по управлению динамически распределяемой памятью. Более глубоко основы и методы управления памятью в Win32 освещены в книге Джеффри Рихтера (Jeffrey Richter) «Programming Applications for Microsoft Windows, Fourth Edition» (Microsoft Press, 1995) (Джеффри Рихтер, «Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows»: «Русская Редакция» — «Питер», М., 2001), в которой рассказано о Windows 2000.

Процессы и адресное пространство

Прежде чем изучать управление памятью в Windows, надо понять, что такое *процесс* (process). Если вы знаете, что такое программа, полдела сделано. Программа — это EXE-файл, который можно запустить из Windows. После запуска программа называется процессом. У процесса есть собственные память, описатели фай-

лов и другие системные ресурсы. Запустив две копии одной программы, вы получите два отдельных процесса. Windows Task Manager (Диспетчер задач Windows) (вызываемый щелчком правой кнопкой мыши панели задач) предоставляет детальный список процессов, выполняемых в данный момент, а также позволяет «убить» (kill), т. е. принудительно завершить процессы, переставшие отвечать на запросы. Программа SPYXX (поставляемая в составе Visual Studio) показывает взаимосвязи между процессами, задачами и окнами.

Примечание Windows Task Manager показывает исполняемые программы и активные процессы. На вкладке Processes (Процессы) показаны активные процессы. У одного процесса (например, Windows Explorer) может быть несколько основных окон, каждое из которых обслуживается отдельным потоком, а у некоторых процессов окна нет вообще (о потоках см. главу 11).

Примечание Microsoft .NET Framework обеспечивает новый, более мелкий уровень деления исполняемого кода — AppDomain, или прикладные домены. С ними мы познакомимся поближе в главе 31.

Важно знать, что процессу выделяется свое «частное» 4-гигабайтное виртуальное адресное пространство, о котором вы подробнее узнаете чуть позже, а пока вообразите, что у вашего компьютера оперативная память размером в сотни гигабайт и каждый процесс исправно получает свои 4 Гб. Программа может обращаться к любому байту этого адресного пространства, используя один-единственный 32-разрядный линейный адрес. При этом в адресном пространстве каждого процесса содержится масса самых разных элементов:

- образ EXE-файла программы;
- все несистемные DLL, загруженные вашей программой (в том числе DLL-модули MFC);
- глобальные данные программы (как доступные для чтения и записи, так и предназначенные только для чтения);
- стек программы;
- динамически выделяемая память, в том числе куча Windows и куча *библиотеки С периода выполнения* (C runtime library, CRT);
- файлы, спроецированные в память;
- блоки памяти, совместно используемые несколькими процессами;
- локальная память отдельных выполняемых потоков;
- всевозможные особые системные блоки памяти, в том числе таблицы виртуальной памяти;
- ядро, исполнительная система и DLL-компоненты Windows.

Адресное пространство процесса в Windows 95/98

В Windows 95 только нижние 2 Гб адресного пространства (0 — 0x7FFFFFFF) по-настоящему закрыты, причем доступ к нижним 4 Мб этих 2 Гб запрещен. Стек, кучи

и глобальная память, доступная для чтения и записи, проецируются на нижние 2 Гб, как, впрочем, и EXE- с DLL-файлами приложения.

Верхние 2 Гб совместно используются всеми процессами. Ядро Windows 95, драйверы виртуальных устройств (VxD), код файловой системы, а также таблицы страниц располагаются в верхнем гигабайте адресного пространства (0xC0000000 — 0xFFFFFFFF). Динамически подключаемые библиотеки и спроецированные в память файлы расположены в диапазоне 0x80000000 — 0xBFFFFFFF. На рис. 10-1 показана схема распределения памяти двух процессов, исполняющих одну и ту же программу.

Насколько все это безопасно? Посторонний процесс практически не в состоянии перезаписать стек, глобальные данные или память кучи другого процесса, потому что вся память в нижних 2 Гб виртуального адресного пространства, принадлежит одному процессу, и только ему. Весь код EXE- и DLL-файлов помечен как доступный только для чтения, поэтому нет никакой проблемы в том, что он спроецирован в несколько процессов.

Однако верхний гигабайт адресного пространства весьма уязвим, поскольку в него спроецированы важные данные Windows, доступные для чтения и записи. При ошибке программа может уничтожить важные системные таблицы, расположенные в этой области. А какой-нибудь процесс может повредить содержимое спроецированных в память (0x80000000 — 0xBFFFFFFF) файлов, потому что эту область совместно используют все процессы.

Адресное пространство Windows NT/2000/XP

Процесс в Windows NT/2000/XP имеет доступ только к нижним 2 Гб своего адресного пространства, причем самые нижние и самые верхние 64 кб этого диапазона недоступны. Исполняемый файл, DLL приложения и Windows, а также спроецированные в память файлы располагаются в диапазоне 0x00010000–0x7FFEFFFF. Ядро, исполнительная система и драйверы устройств Windows NT располагаются в верхних 2 Гб, где они полностью защищены от искажения со стороны неисправной программы. Файлы, спроецированные в память, также реализованы надежнее: никакой процесс не получит доступа к спроецированному в память файлу другого процесса, если только он не знает имени файла и не создал проекцию явно.

Устройство виртуальной памяти

Конечно, на самом деле никаких сотен гигабайт оперативной памяти в вашем компьютере нет. Нет и сотен гигабайт дискового пространства. Здесь Windows показывает «ловкость рук» настоящего фокусника.

Во-первых, 4-гигабайтное адресное пространство процесса экономно используется небольшими фрагментами. Программы и элементы данных разбросаны по адресному пространству блоками по 4 кб, выровненными по границам, кратным 4 кб. Каждый такой блок называется *страницей* (page) и содержит либо код, либо данные. Естественно, используемая страница занимает какой-то участок физической памяти, но вы никогда не узнаете ее физического адреса. Микропроцессор фирмы Intel эффективно преобразует 32-разрядный виртуальный адрес в номер

физической страницы и смещение внутри нее, пользуясь двухуровневыми таблицами 4-килобайтных страниц (рис. 10-2). Заметьте: отдельные страницы можно помечать либо как «только для чтения», либо как «для чтения и записи». Кроме того,

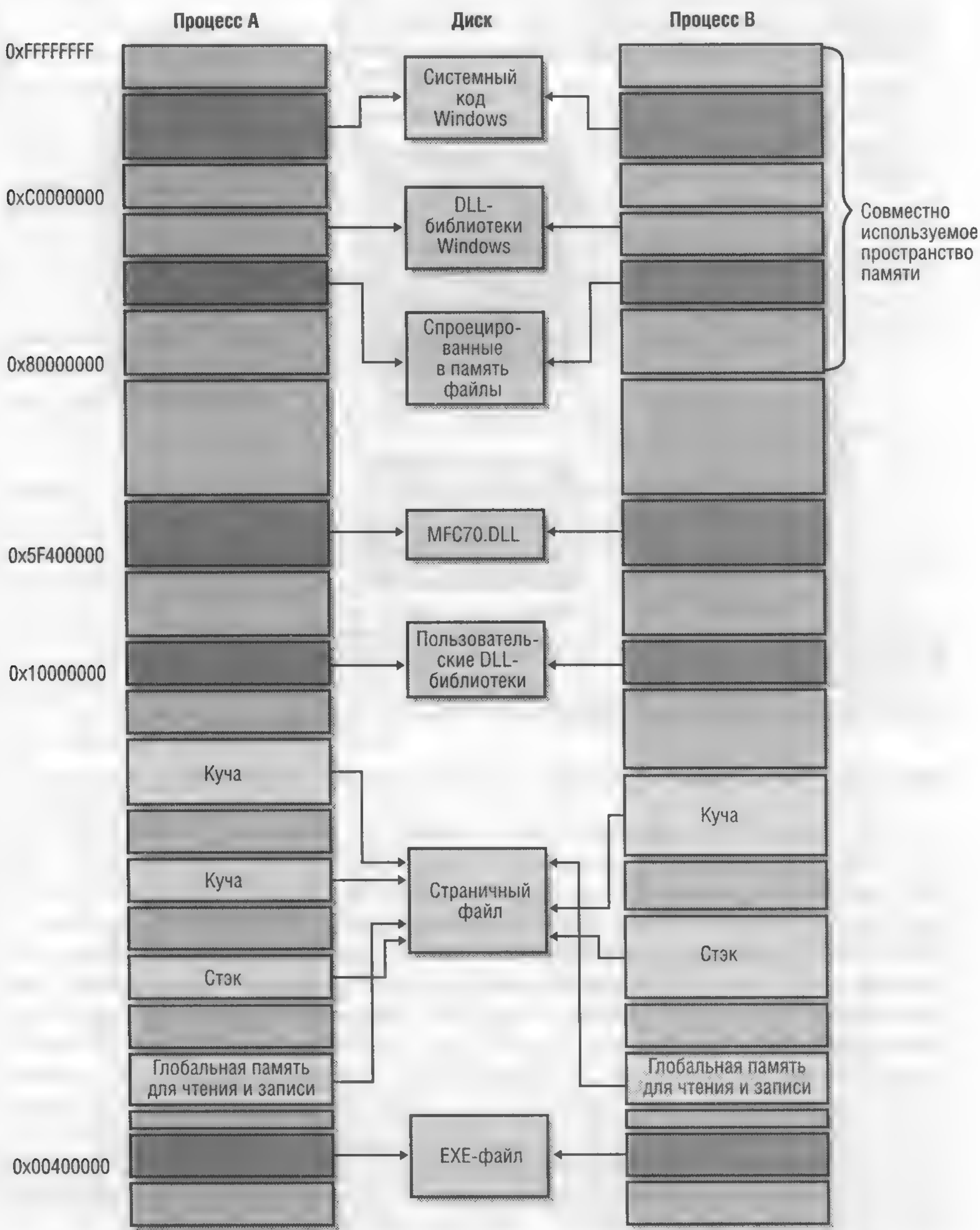


Рис. 10-1. *Типичное распределение виртуальной памяти двух процессов, связанных с одним и тем же EXE-файлом в Windows 95/98*

у каждого процесса свой набор таблиц страниц. Регистр CR3 процессора содержит указатель на страницу каталога, поэтому при переключении с одного процесса на другой Windows просто обновляет этот регистр.

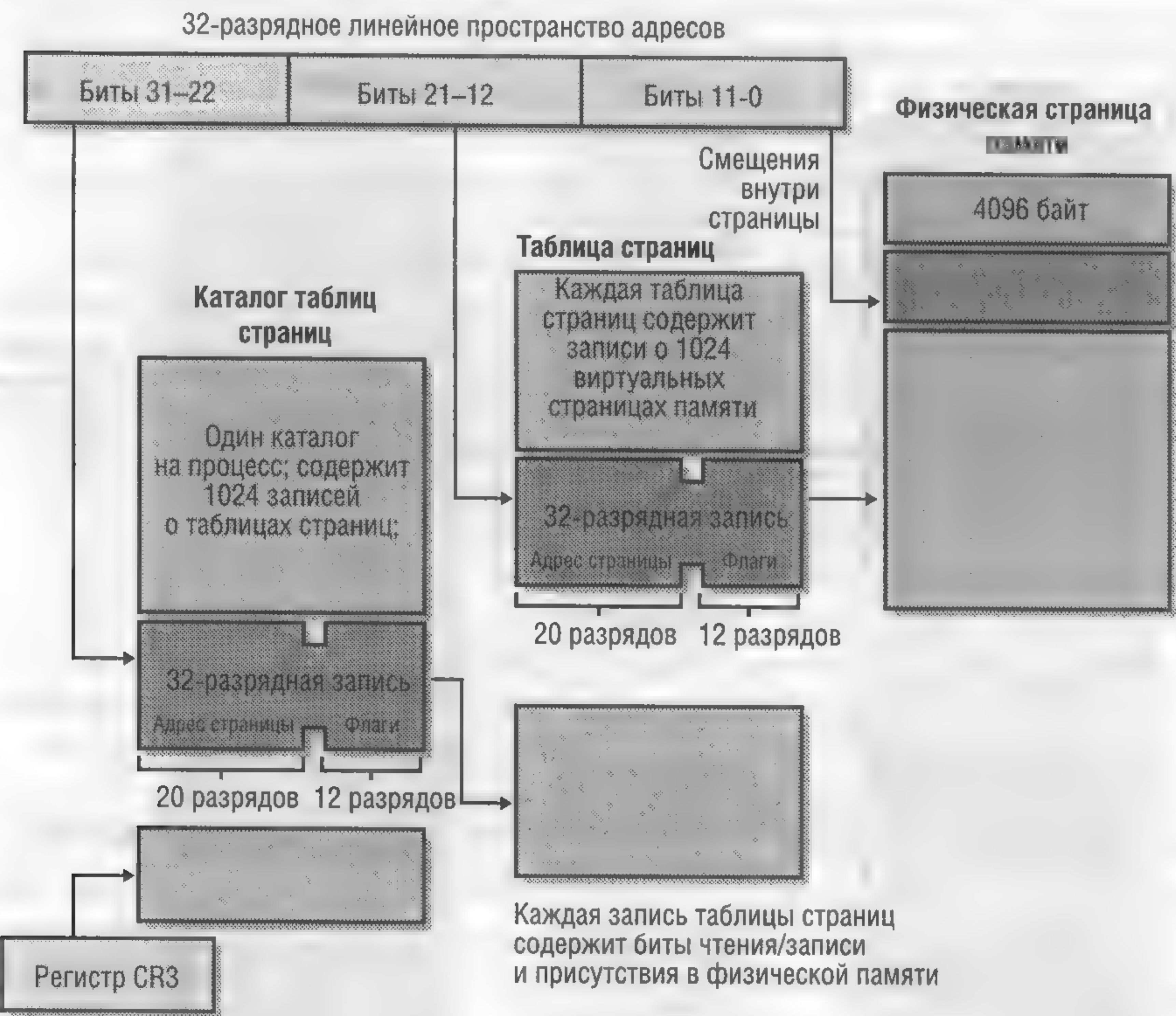


Рис. 10-2. *Управление виртуальной памятью в Win32 (на процессоре Intel)*

Итак, теперь объем пространства, занимаемого нашим процессом, сократился с 4 Гб до, скажем, 5 Мб — прогресс налицо. Но если мы запустим несколько программ (помимо самой Windows!), нам опять не хватит оперативной памяти. Еще раз обратившись к рис. 10-2, вы заметите, что запись таблицы страниц содержит бит присутствия в физической памяти (present), который сообщает, находится ли сейчас в физической памяти данная 4-килобайтная страница. При попытке обращения к странице, отсутствующей в памяти, инициируется прерывание, и Windows анализирует ситуацию, просматривая свои внутренние таблицы. Если ссылка на область памяти оказывается ошибочной, мы получим ненавистное сообщение об *ошибке страницы* (page fault), и программа завершится. В противном случае Windows считает в оперативную память нужную страницу из дискового файла и обновит таблицу страниц, записав в нее физический адрес и установив бит присутствия в физической памяти. Так работает виртуальная память в Win32.

Диспетчер виртуальной памяти Windows, стремясь достичь максимума производительности, определяет моменты чтения и записи страниц. Если какой-то

процесс не использовал страницу в течение определенного периода, а другому нужна память, страница выгружается из памяти, а вместо нее загружается страница нового процесса. Обычно программа об этом не уведомляется. Но вы должны понимать: чем интенсивнее дисковый ввод/вывод, тем ниже производительность, вот почему всегда желательно иметь побольше оперативной памяти.

Мы употребили слово «диск», но еще ничего не сказали о файлах. Все процессы совместно используют один большой общесистемный *страничный файл* (swap file) (ранее он назывался *файл подкачки*), куда помещаются (при необходимости) все виды данных для чтения и записи и некоторые виды данных только для чтения. (Windows NT/200/XP способна одновременно поддерживать несколько страничных файлов.) Windows определяет размер страничного файла в зависимости от объема ОЗУ и свободного дискового пространства, но есть способы тонкой настройки размера и физического расположения этого файла.

Диспетчер виртуальной памяти, однако, использует не только страничный файл . Записывать в него страницы кода резона нет. Вместо этого Windows проецирует содержимое EXE- и DLL-модулей прямо в их дисковые файлы. Поскольку страницы кода помечены «только для чтения», необходимости в их записи обратно на диск не возникает.

Если два процесса используют один и тот же EXE-файл, последний отображается на адресные пространства обоих процессов. Код и константы никогда не изменяются во время выполнения программы, поэтому можно проецировать файл на одну и ту же область физической памяти. Однако два процесса не могут совместно обращаться к глобальным данным. С ними Windows 95/98 и Windows NT/200/XP поступают по-разному. Windows 95/98 проецируют в каждый процесс отдельную копию глобальных данных, а вот в Windows NT/200/XP оба процесса используют одну копию глобальных данных до тех пор, пока один из процессов не попытается что-либо записать в страницу. В этот момент страница копируется, а затем у каждого процесса появляется собственная копия, находящаяся по одинаковому виртуальному адресу.

Примечание Динамически подключаемая библиотека проецируется непосредственно на свой файл DLL, только если DLL-модуль удается загрузить по заданному базовому адресу. Если DLL статически скомпонована для загрузки по адресу, скажем, 0x10000000, а этот диапазон адресов уже занят другой DLL, Windows нужно «урегулировать» адреса в самом коде DLL. Windows NT/200/XP копируют модифицированные страницы в страничный файл при первичной загрузке DLL, а Windows 95/98 выполняют настройку адресов «на лету» при копировании страниц в память. Стоить ли говорить о том очевидном факте, что в ваших DLL диапазоны адресов не должны перекрываться? Если вы применяете DLL-модули MFC, установите базовый адрес своих DLL вне диапазона 0x5F400000–0x5FFFFFFF. Подробнее о том, как писать DLL, мы поговорим в главе 20.

Файлы, проецируемые в память, которые мы обсудим позже, также отображаются напрямую. Их можно определить как доступные для чтения и записи или совместно используемые несколькими процессами.

Для тех, кто программирует в Win32

Если вы экспериментировали с окном Registers в Visual Studio, то наверняка обратили внимание на *сегментные регистры* (segment registers), в частности CS, DS и SS. (Чтобы увидеть сегментные регистры в Visual Studio .NET, нужно щелкнуть правой кнопкой в окне Registers и в контекстном меню выбрать группу CPU Segments). Да-да, эти 16-разрядные реликты еще не исчезли, но обычно их можно игнорировать. В 32-разрядном режиме микропроцессор Intel — для преобразования адресов, прежде чем передать их в систему виртуальной памяти, — по-прежнему использует 16-разрядные сегментные регистры. В оперативной памяти хранится *таблица дескрипторов* (descriptor table), элементы которой содержат базовые адреса виртуальной памяти и размеры блоков для сегментов кода, данных и стека. В 32-разрядном режиме размер этих сегментов может достигать 4 Гб; их можно помечать как «только для чтения» или «для чтения и записи». При каждом обращении к памяти микропроцессор использует селектор для поиска нужной записи в таблице дескрипторов и транслирует адрес.

В Win32 у каждого процесса два сегмента: один для кода, а второй для данных и стека. У обоих базовый адрес 0 и размер 4 Гб, т. е. эти сегменты перекрываются. В результате трансляция адресов становится ненужной, хотя парочку трюков, чтобы исключить из сегмента данных нижние 16 кб, Windows все же применяет. Обратившись к этим, самым младшим адресам, вы получите ошибку защиты, а не ошибку страницы, что весьма полезно, когда при отладке возникают нулевые указатели.

Придет время, и какая-то ОС будущего с помощью сегментов и обойдет это 4-гигабайтное ограничение.

Функция *VirtualAlloc*: переданная и зарезервированная память

Если вашей программе нужна динамически распределяемая память, то рано или поздно придется вызвать Win32-функцию *VirtualAlloc*. Скорее всего делать это будете не вы, а функции Windows или библиотеки С периода выполнения, выделяющие память из кучи. Но зная, как работает *VirtualAlloc*, вы лучше поймете функции, которые к ней обращаются.

Сначала разберемся с понятиями *зарезервированная* (reserved) и *переданная* (committed) память. При резервировании памяти выделяется непрерывный диапазон виртуальных адресов. Если вы, допустим, знаете, что программа будет оперировать с одной 5-мегабайтной областью памяти [области памяти иногда называют *регионами* (regions)], но вся она вам сейчас не нужна, вызовите *VirtualAlloc* и в параметре, определяющем тип выделения памяти, укажите *MEM_RESERVE*, а в параметре, задающем размер выделяемой памяти, — 5 Мб. Windows округляет начальный и конечный адреса области до значений, кратных 64 кб, и уже не даст вашему процессу повторно зарезервировать память из этой области. И, хотя можно указать начальный адрес области, лучше оставить это занятие самой Windows.

Ну вот, собственно, и все на этом этапе. Больше ничего не происходит: не выделяются ни оперативная памяти, ни пространство в страничном файле.

Когда вам действительно понадобится оперативная память, вы снова вызовете *VirtualAlloc*, указав на этот раз *MEM_COMMIT*, чтобы передать память из этой области. Теперь начальный и конечный адреса блока округляются до значений, кратных 4 кб, и в страничном файле выделяются соответствующие страницы, а также создается нужная таблица страниц. Блок помечается либо «только для чтения», либо «для чтения и записи». Однако физическая память по-прежнему не выделяется — это произойдет, лишь когда вы попытаетесь получить доступ к этому блоку памяти. Ничего страшного, если передаваемая память ранее не была зарезервирована или была передана — главное, что перед использованием память следует передать.

Для возвращения (decommit) переданной память (по сути возврата соответствующим страницам статуса зарезервированных) применяется функция *VirtualFree*. Она может также освободить зарезервированную область памяти, но для этого ей надо передать базовый адрес, возвращенный предыдущим вызовом *VirtualAlloc* при резервировании памяти.

Куча Windows и семейство функций *GlobalAlloc*

Куча (heap) — это пул памяти какого-либо процесса. Когда программе нужен блок памяти, вы вызываете функцию, выделяющую память из кучи, а чтобы освободить память — функцию, выполняющую обратное. Выравнивания по границам, кратным 4 кб, при этом не происходит — диспетчер кучи использует пространство на выделенных страницах или обращается к *VirtualAlloc* за дополнительными страницами. Сначала мы познакомимся с кучами Windows, а затем — с кучами, управляемыми библиотекой С периода выполнения, в частности функциями *malloc* и *new*.

Windows предоставляет каждому процессу кучу по умолчанию, а процесс может создать любое число дополнительных куч Windows. Для выделения памяти из куч Windows служит функция *HeapAlloc*, а для освобождения — *HeapFree*.

Скорее всего вы не будете вызывать *HeapAlloc* — за вас это сделает унаследованная от Win16 функция *GlobalAlloc*. В идеальном мире 32-разрядных программ вам не понадобилась бы *GlobalAlloc*, но мы живем в реальном мире. У нас все еще остается колоссальный объем кода, перенесенного из Win16, в котором вместо 32-разрядных адресов применяются параметры типа «описатель памяти» (*HGLOBAL*).

GlobalAlloc использует кучу Windows по умолчанию. Работа этой функции зависит от передаваемых ей атрибутов. Если указать *GMEM_FIXED*, она просто вызывает *HeapAlloc* и возвращает адрес, приводя его к 32-разрядному значению *HGLOBAL*. Если же вы передаете *GMEM_MOVEABLE*, возвращаемое значение *HGLOBAL* является указателем на элемент таблицы описателей данного процесса. В этом элементе содержится указатель на память, выделенную функцией *HeapAlloc*.

Зачем нужна *перемещаемая* (moveable) память, если она вводит в управление памятью еще один промежуточный слой? Здесь мы встречаемся с наследием Win16, где ОС действительно перемещала блоки памяти. В Win32 перемещаемые блоки памяти существуют лишь для поддержки *GlobalReAlloc*, которая выделяет новый блок памяти, копирует в него содержимое старого блока, освобождает последний и по-

мещает адрес нового блока в существующий элемент таблицы описателей. Если бы никто не вызывал *GlobalReAlloc*, мы могли бы обойтись *HeapAlloc* вместо *GlobalAlloc*[1].

К сожалению, многие библиотечные функции используют в качестве параметров и возвращаемых значений *HGLOBAL*, а не адреса памяти. Если такая функция возвращает *HGLOBAL*, вы должны считать, что память выделена с атрибутом *GMEM_MOVEABLE*, и, следовательно, чтобы получить адрес памяти, надо вызвать функцию *GlobalLock*. (В случае фиксированной памяти *GlobalLock* просто возвращает переданный ей описатель как адрес.) По завершении работы с памятью ее освобождают вызовом *GlobalUnlock*. Если вы должны передать параметр *HGLOBAL*, то для верности следует получить это значение с помощью *GlobalAlloc(GMEM_MOVEABLE,...)* — на случай, если вызываемая функция обращается к *GlobalReAlloc* и ожидает, что значение описателя не изменится.

Куча малых блоков, *_heapmin* и операторы *new* и *delete* в C++

Вы вправе применять *HeapAlloc*, но скорее всего вы будете работать с функциями *malloc* и *free* библиотеки C периода выполнения. Если же вы программируете на C++, то будете работать с операторами *new* и *delete*, напрямую вызывающими *malloc* и *free*. Если с помощью *new* выделяется блок, превышающий определенный предел (по умолчанию 480 байт), то CRT (C++ runtime library) сразу передает вызов функции *HeapAlloc* — для выделения памяти из кучи Windows, созданной для CRT. Блоками, меньшими указанного предела, управляет сама CRT с помощью *кучи малых блоков* (small-block heap), вызывая при необходимости *VirtualAlloc* и *VirtualFree*. Алгоритм работы таков.

1. Память резервируется областями по 4 Мб.
2. Память передается блоками по 64 кб (16 страниц).
3. Память возвращается по 64 кб за раз. Если освобождается 128 кб, то возвращаются последние 64 кб.
4. 4-мегабайтная область освобождается, когда возвращены (decommitted) все ее страницы.

Как видите, куча малых блоков сама занимается очисткой. А вот куча Windows автоматически не возвращает страницы и не отказывается (unreserve) от них. Для очистки больших блоков нужна CRT-функция *_heapmin*, которая вызывает функцию Windows *HeapCompact*. (Увы, версия *HeapCompact* для Windows 95/98 не делает ничего — еще один довод в пользу Windows NT/2000/XP.) Как только страницы возвращены, другие программы могут обратиться к освободившемуся пространству страничного файла.

---

[1] Очень спорное утверждение. Во-первых, для копирования данных в *буфер обмена* (clipboard) нужно выделить блок памяти вызовом *GlobalAlloc* с атрибутами *GMEM_MOVEABLE* и *GMEM_DDESHARE*. Во-вторых, выделение памяти с атрибутом *GMEM_MOVEABLE* позволяет системе перемещать блоки памяти внутри кучи, тем самым не допуская ее фрагментации. — *Прим. перев.*

Примечание В предыдущих версиях CRT указатели списка свободных блоков памяти хранились в страницах самой кучи. Такой подход требовал от *malloc* в поисках свободного пространства «перелистывать» (считывать из страничного файла) множество страниц, что снижало производительность. Современная система, хранящая список свободных блоков в отдельной области памяти, работает быстрее и уменьшает необходимость в дополнительном ПО для управления кучей.

Если вы хотите узнать или изменить предельное значение размера блока, используйте CRT-функции *_set_sbh_threshold* и *_get_sbh_threshold*.

Специальная отладочная версия *malloc* — *_malloc_dbg* — записывает в выделяемые блоки памяти отладочную информацию. *_malloc_dbg* вызывается оператором *new*, когда вы собираете MFC-проект с определенным символом *_DEBUG*. При этом программа может обнаружить блоки памяти, которые вы забыли освободить или нечаянно затерли.

Проецируемые в память файлы

Если вы думаете, что у вас недостаточно возможностей управления памятью, рассмотрим еще один вариант. Допустим, программа должна считывать DIB-файл (Device-Independent Bitmap — растровое изображение в аппаратно-независимом формате). Очевидное решение — выделить буфер нужного размера, открыть файл и вызвать функцию чтения, чтобы целиком скопировать дисковый файл в буфер. Но куда элегантнее спроецировать файл в память, т. е. просто отобразить диапазон адресов прямо на файл. Когда процесс обращается к какой-то странице памяти, Windows выделяет область оперативной памяти и считывает данные с диска. Код выглядит примерно так:

```
HANDLE hFile = ::CreateFile(strPathname, GENERIC_READ,
    FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
ASSERT(hFile != NULL);
HANDLE hMap = ::CreateFileMapping(hFile, NULL, PAGE_READONLY,
    0, 0, NULL);
ASSERT(hMap != NULL);
LPVOID lpvFile = ::MapViewOfFile(hMap, FILE_MAP_READ,
    0, 0, 0); // Проецируем целый файл
DWORD dwFileSize = ::GetFileSize(hFile, NULL);  // полезная  информация
// Файл используется
::UnmapViewOfFile(lpvFile);
::CloseHandle(hMap);
::CloseHandle(hFile);
```

Здесь используется виртуальная память, содержимое которой хранится в DIB-файле. Windows определяет размер файла и передает соответствующую область виртуальной памяти. В данном случае ее начальный адрес — *lpvFile*. Переменная *hMap* содержит описатель объекта «проекция файла», к которому могут совместно обращаться несколько процессов.

DIB-файл в этом примере невелик, поэтому его можно полностью считать в буфер. Но представьте себе файл большего размера, с которым обычно работают

с применением *команд поиска* (seek). Файлы, проецируемые в память, будут работать и с таким файлом благодаря системе виртуальной памяти, лежащей в их основе. Память выделяется, и страницы читаются, только когда к ним обращаются, и не раньше.

Примечание По умолчанию файл проецируется целиком, хотя можно проецировать лишь его часть.

Если два процесса совместно обращаются к объекту «проекция файла» (скажем, *hMap*, как в предыдущем примере), то файл по сути становится совместно используемой памятью; но виртуальные адреса, возвращаемые *MapViewOfFile*, могут различаться. Именно таков предпочтительный способ совместного использования памяти в Win32. (Вызов функции *GlobalAlloc* с флагом *GMEM_SHARE* не создает совместно используемую память, как было в Win16.) Если вам нужно только совместное использование памяти, а постоянный дисковый файл не требуется, можете не вызывать *CreateFile*, а передать функции *CreateFileMapping* в параметре *hFile* значение 0xFFFFFFFF. Тогда содержимое совместно используемой памяти будет храниться в страницах страничного файла. (Подробности — в книге Джеффри Рихтера).

Примечание Если вам нужен произвольный доступ к небольшому числу страниц спроецированного в память файла, основанного на страничном файле, используйте прием, описанный Джеффри Рихтером. В этом случае *CreateFileMapping* вызывается со специальным флагом, а затем с помощью *VirtualAlloc* передаются только заданные диапазоны адресов.

Примечание Обратите внимание на сообщение Windows *WM_COPYDATA*. Оно позволяет процессам обмениваться данными через совместно используемую память, не обращаясь к API-функциям, применяемых для работы с проецируемыми файлами. Это сообщение является синхронным, т. е. процесс-отправитель должен подождать, пока процесс-получатель скопирует и обработает данные.

К сожалению, MFC не поддерживает проецирование файлов или совместное использование памяти напрямую. Класс *CSharedFile* поддерживает только обмен данными через *буфер обмена* (clipboard) при помощи описателей HGLOBAL, так что этот класс не настолько универсален, как может показаться из названия.

Доступ к ресурсам

Ресурсы содержатся в EXE- или DLL-файлах и, таким образом, занимают виртуальное адресное пространство, содержимое которого не меняется в течение жизни процесса, поэтому ресурсы легко читать напрямую. Если вам, допустим, нужен доступ к растровому изображению, адрес DIB позволяет получить примерно такой код:

```
LPVOID lpvResource = (LPVOID) ::LoadResource(NULL,
    ::FindResource(NULL, MAKEINTRESOURCE(IDB_REDBLOCKS), RT_BITMAP));
```

Функция *LoadResource* возвращает значение HGLOBAL, но его можно безопасно приводить к указателю.

Советы по работе с кучей

Чем интенсивнее используется куча, тем больше она фрагментируется и тем медленнее работает программа. Если время непрерывного функционирования программы будет исчисляться часами или днями, следует проявить особую осторожность. Лучше выделить всю нужную память при запуске программы и освободить при закрытии, но это не всегда возможно. Тут может помешать класс *CString*, который постоянно выделяет и освобождает крошечные порции памяти.

Не забывайте вызывать *_heapmin* каждый раз, когда программа выделяет блоки размера, превышающего верхний предел для кучи малых блоков. Следите за тем, откуда получена динамически распределяемая память: у вас возникнут жуткие проблемы, если, скажем, вы вызовете *HeapFree*, передав ей указатель на малый блок, полученный от *new*.

Учтите: размер стека может быть практически любым. Так как ограничения в 64 кб теперь нет, в стек можно помещать объекты большого размера, что уменьшает необходимость в распределении памяти из кучи.

Ваша программа, работая на полной скорости, не сгенерирует исключение, если Windows не хватит страничного файла. Она просто станет постепенно замедляться и в конце концов остановится, что вряд ли понравится пользователю. Здесь от вас мало что зависит — можно лишь пытаться определить, какая программа пожирает память и почему. Поскольку в модулях GDI и USER Windows 95/98 по-прежнему есть 16-разрядные компоненты, сохраняется вероятность переполнения 64-килобайтных куч, хранящих объекты GDI и оконные структуры. Однако такая вероятность крайне мала, и если дела обстоят именно так, то ошибка скорее всего в вашей программе.

Оптимизация хранения констант

Вспомните, что код вашей программы хранится не в страничном файле, а прямо в EXE- и DLL-файлах. Если запущено сразу несколько экземпляров программы, на виртуальные адресные пространства каждого процесса будут спроецированы те же EXE и DLL. Но что при этом происходит с константами? Предпочтительнее, чтобы они были частью программы, а не копировались в другой блок виртуальной памяти, хранимый в страничном файле.

Чтобы константы гарантированно хранились вместе с программой, придется поработать. Во-первых, обратите внимание на строковые константы, которыми часто изобилуют программы. Может, вы решили, что они будут данными «только для чтения»? Не совсем — пошевелите-ка еще раз извилинами. Действительно, вы имеете право написать что-то вроде:

```
char* pch = "test";
*pch = 'x';
```

Но *test* не может быть константой. Чтобы строка стала константой, ее надо соответственно объявить и инициализировать, скажем, так:

```
const char g_pch[] = "test";
```

Теперь *g_pch* хранится вместе с кодом. Но где именно? Чтобы ответить на этот вопрос, надо знать о *секциях данных* (data sections), генерируемых компоновщиком Visual C++. Если потребовать от компоновщика, чтобы он генерировал *файл компоновки* (map file), то в последнем вы найдете длинный список секций (блоков памяти) вашей программы. Отдельные секции могут предназначаться для хранения кода или данных, а также помечаться «только для чтения» или «для чтения и записи». Вот основные секции и их характеристики:

Табл. 10-1. Важные секции программы

Имя	Тип	Доступ	Содержимое
.text	Код	Только чтение	Код программы
.rdata	Данные	Только чтение	Инициализированные константы
.data	Данные	Чтение и запись	Инициализированные данные (не константы)
.bss	Данные	Чтение и запись	Неинициализированные данные (не константы)

Секция .rdata — часть EXE-файла; именно сюда компоновщик поместит переменную *g_pch*. Чем больше данных вы поместите сюда, тем лучше. Для этого служит модификатор *const*. В секцию .rdata можно включить встроенные типы и даже структуры, но не объекты C++, у которых есть конструктор. Если написать оператор:

```
const CRect g_rect(0, 0, 100, 100);
```

компоновщик внесет этот объект в секцию .bss, а в страничном файле будет выделено место для хранения отдельной копии объекта для каждого процесса. Если хорошенько подумать, то так и должно быть, потому что компилятор вызывает функцию-конструктор после загрузки программы.

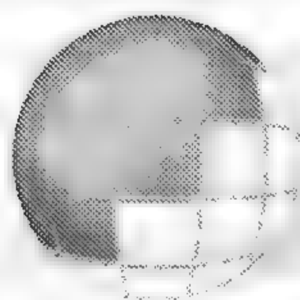
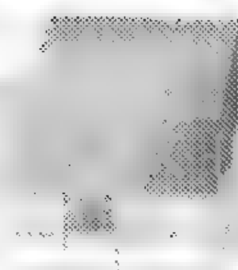
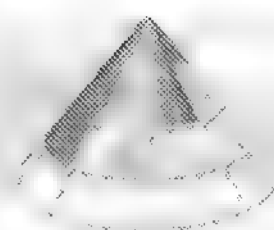
А теперь допустим, что вы собираетесь сделать худшее — объявить глобальную переменную (или статическую переменную-член класса) типа *CString* вроде:

```
const CString g_str("this is the worst thing I can do"); // хуже не придумаешь
```

Вы получите объект *CString*, размер которого весьма мал, в секции .bss и массив символов в секции .data. Ни тот, ни другой не могут храниться в EXE-файле в период выполнения. Хуже того, при запуске программы класс *CString* должен выделить память из кучи для хранения копии массива символов. Так что в данном случае вместо объекта *CString* лучше задействовать *const*-массив символов.

ГЛАВА

11

Обработка сообщений Windows и многопоточные приложения

Вытесняющая многозадачность и многопоточность в Win32 произвели настоящую революцию в программировании для Windows. Если вам попадались статьи или книги по этим вопросам, вы наверняка уже оценили всю сложность применения многих потоков. Вообще-то вы еще долго сможете разрабатывать полезные однопоточные Windows-приложения. Но изучив основы многопоточности, вы сможете создавать более эффективные и мощные программы и лучше разберетесь в модели программирования Win32.

Обработка сообщений Windows

Чтобы разобраться в потоках, сначала надо понять, как 32-разрядная Windows обрабатывает сообщения. Лучше всего начать с однопоточной программы, которая продемонстрирует значимость процесса преобразования и рассылки сообщений. Затем мы усовершенствуем ее, добавив поток, управляемый с помощью глобальной переменной и простого сообщения. Потом поэкспериментируем с событиями и критическими секциями. А чтобы вникнуть в более сложные элементы, обеспечивающие многопоточность (такие как мьютексы и семафоры), вам придется обратиться к другой книге, например, к уже упоминавшемуся труду Джеффри Рихтера.

Обработка сообщений в однопоточной программе

До сих пор мы писали только *однопоточные* (single-threaded) программы, т. е. у кода был лишь один поток исполнения. Используя мастер Microsoft Visual Studio,

вы создавали функции-обработчики различных сообщений Windows, а также писали код функции *OnDraw*, вызываемой в ответ на сообщение *WM_PAINT*. Казалось бы, при появлении сообщения каким-то чудом вызывается соответствующий обработчик, но все не так просто. Глубоко в недрах MFC-кода, компонуемого с вашей программой, спрятаны примерно такие инструкции:

```
MSG message;
while (::GetMessage(&message, NULL, 0, 0)) {
    ::TranslateMessage(&message);
    ::DispatchMessage(&message);
}
```

Windows определяет, какие сообщения принадлежат вашей программе, а функция *GetMessage* возвращает управление, как только появляется сообщение для обработки. Если сообщений нет, ваша программа приостанавливается, и выполняются другие приложения. Когда сообщение наконец поступает, ваша программа «пробуждается». Функция *TranslateMessage* преобразует сообщения *WM_KEYDOWN* в сообщения *WM_CHAR*, содержащие ASCII-символы, а *DispatchMessage* передает управление (через оконный класс) коду выборки сообщений MFC, который вызывает вашу функцию на основе таблицы обработчиков сообщений. Завершив работу, обработчик возвращает управление MFC-коду, что в итоге вызывает возврат из *DispatchMessage*.

Передача управления

А что, если одна из ваших функций-обработчиков окажется «свиньей», алчущей процессорных ресурсов, и израсходует 10 секунд процессорного времени? Во времена 16-разрядной системы компьютер просто завис бы на это время. Доступными остались бы только перемещение курсора мыши да пара-тройка других задач, управляемых прерываниями. В Win32 многозадачность организована куда лучше. Вытесняющая многозадачность не даст другим приложениям зависнуть: Windows, когда сочтет это нужным, просто приостановит выполнение «жадной» функции. Однако даже в Win32 на эти 10 секунд программа будет заблокирована. Она не сможет обрабатывать сообщения, так как *DispatchMessage* не возвратит управление, пока его не возвратит злополучный обработчик.

Однако обойти эту проблему можно, причем как в Win16, так и в Win32. Надо просто заставить «жадину» вести себя дружелюбнее, т. е. периодически отдавать управление — а для этого нужно вставить в основной цикл такой функции операторы:

```
MSG message;
if (::PeekMessage(&message, NULL, 0, 0, PM_REMOVE)) {
    ::TranslateMessage(&message);
    ::DispatchMessage(&message);
}
```

Функция *PeekMessage* работает так же, как и *GetMessage* за исключением того, что возвращает управление сразу, даже в отсутствие соответствующих сообщений. При этом «свинская» функция продолжает поглощать процессорное время. Но стоит

появиться сообщению, вызывается обработчик, и по завершении его работы выполнение функции возобновляется.

Таймеры

Таймер Windows — это полезный элемент, иногда устраняющий необходимость в многопоточном программировании. Например, если вам нужно считывать содержимое коммуникационного буфера, установите таймер так, чтобы выбирать накопившиеся символы каждые 100 мс. Таймер можно применять и для управления анимацией, поскольку он не зависит от тактовой частоты процессора.

Работать с таймерами легко. Вы просто вызываете функцию *CWnd::SetTimer* с параметром — интервалом времени, после чего определяете обработчик сообщения *WM_TIMER*. После запуска таймера с заданным интервалом в миллисекундах сообщения *WM_TIMER* постоянно посылаются вашему окну, пока не будет вызвана *CWnd::KillTimer* или уничтожено окно. Можно задействовать и несколько таймеров, каждый из которых идентифицируется целым числом. Поскольку Windows не является ОС реального времени, точность соблюдения интервала длительностью, значительно меньшей 100 мс, будет невысока.

Как и другие сообщения Windows, сообщения таймера могут заблокировать другие функции-обработчики в вашей программе. К счастью, сообщения таймера не аккумулируются. Windows не ставит сообщения таймера в очередь, если в ней уже есть одно сообщение от данного таймера.

Пример Ex11a

Мы напишем однопоточную программу с циклом, в котором выполняется большой объем вычислений. Программа должна обрабатывать сообщения после того, как пользователь приступит к вычислениям, иначе он не сможет прервать их. Кроме того, нам хотелось бы отображать процент выполненной работы, применив элемент управления «индикатор хода процесса» (рис. 11-1). Программа Ex11a обеспечивает обработку сообщений, передавая управление в цикле вычислений. Обработчик таймера обновляет индикатор в соответствии с продвижением процесса вычислений. Если бы процесс вычислений не передавал управление, нам не удалось бы обработать сообщения *WM_TIMER*.

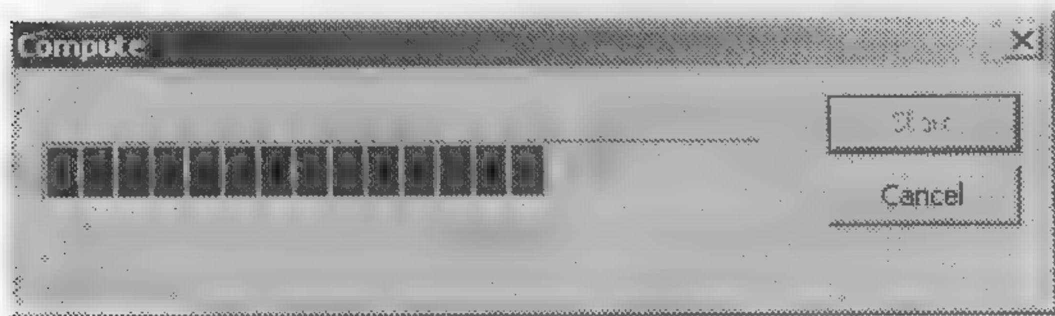


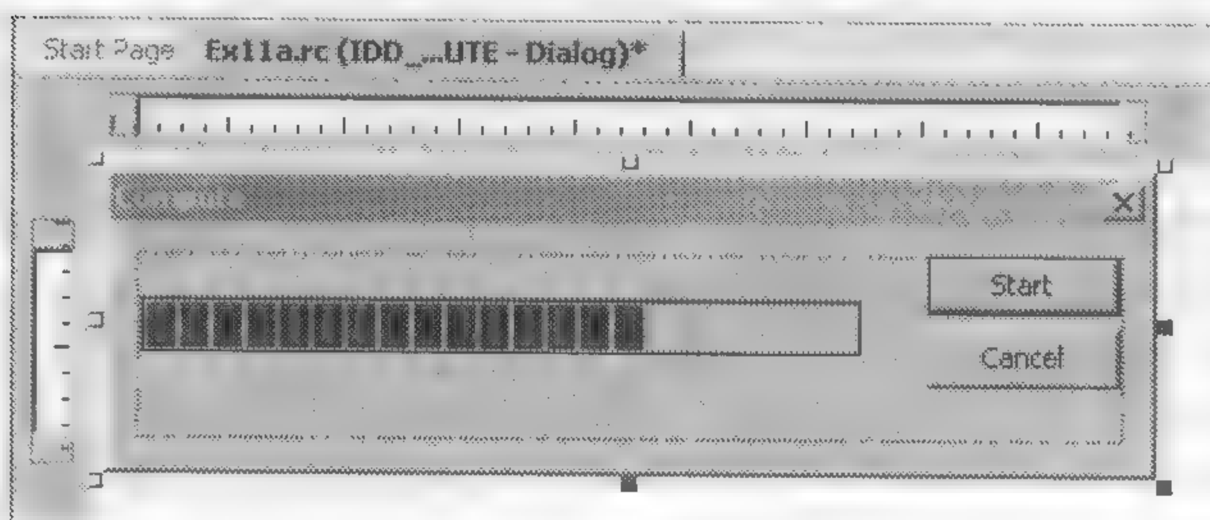
Рис. 11–1. *Диалоговое окно Compute*

Итак, разработаем приложение Ex11a.

1. **Используя MFC Application Wizard, создайте проект Ex11a.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC Application, а в качестве имени — Ex11a. На странице Application Type мастера установите переключатель в положение Single document, а

на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.

2. **В редакторе диалоговых окон создайте диалоговый ресурс *IDD_COMPUTE*.** Выберите в меню Project среды разработки команду Add Resource и в открывшемся одноименном диалоговом окне щелкните строку Dialog, а затем — кнопку New. Visual Studio создаст новый диалоговый ресурс. Измените идентификатор ресурса на *IDD_COMPUTE*, а свойство Caption — на Compute. Измените идентификатор кнопки OK на *IDC_START*, а свойство Caption — на Start. У кнопки Cancel измените идентификатор на *IDC_CANCEL*. Через панель инструментов Toolbox добавьте в окно индикатор хода процесса (Progress Control) и оставьте идентификатор по умолчанию *IDC_PROGRESS1*. По завершении этих операций диалоговое окно должно выглядеть так:



3. **Средствами MFC Class Wizard создайте класс *CComputeDlg*.** В меню Project выберите команду Add Class и в открывшемся окне MFC Class Wizard введите имя класса *CComputeDlg*, в качестве базового класса выберите *CDialog* и в поле Dialog ID выберите *IDD_COMPUTE*, чтобы связать новый класс с созданным диалоговым ресурсом.
4. **Создайте обработчики сообщений *WM_TIMER* и *BN_CLICKED*.** Выберите класс *CComputeDlg* в Class View, в окне Properties щелкните кнопку Messages и добавьте функцию *OnTimer* для сообщения *WM_TIMER*. Щелкните кнопку Events и добавьте функции *OnBnClickedStart* и *OnBnClickedCancel* для *IDC_START* и *IDC_CANCEL*
5. **Добавьте три переменных-члена в класс *CComputeDlg*.** Отредактируйте файл ComputeDlg.h, добавив закрытые переменные-члены:

```
private:
    int m_nTimer;
    int m_nCount;
    enum { nMaxCount = 50000 };
```

Переменная-член *m_nCount* класса *CComputeDlg* будет увеличиваться в процессе выполнения вычислений. Деление ее на константу *nMaxCount* даст единицу измерения продвижения.

6. **Добавьте код инициализации в конструктор *CComputeDlg* в файле ComputeDlg.cpp.** Напишите в конструкторе следующую строку (чтобы кнопка Cancel действовала, если вычисления еще не начаты):

```
m_nCount = 0;
```

7. **Запрограммируйте функцию *OnBnClickedStart* в файле ComputeDlg.cpp.** Этот код исполняется, когда пользователь щелкает кнопку Start. Добавьте выделенный код:

```
void CComputeDlg::OnBnClickedStart()
{
   MSG message;

   m_nTimer = SetTimer(1, 100, NULL); // 1/10 секунды
   ASSERT(m_nTimer != 0);
   GetDlgItem(IDC_START)->EnableWindow(FALSE);
   volatile int nTemp;
   for (m_nCount = 0; m_nCount < nMaxCount; m_nCount++) {
      for (nTemp = 0; nTemp < 10000; nTemp++) {
         // uses up CPU cycles
      }
      if (::PeekMessage(&message, NULL, 0, 0, PM_REMOVE)) {
         ::TranslateMessage(&message);
         ::DispatchMessage(&message);
      }
   }
   GetDlgItem(IDC_START)->EnableWindow(TRUE);
   CDialog::OnOK();
}
```

Основной цикл *for* контролируется счетчиком *m_nCount*. На каждой итерации цикла *PeekMessage* позволяет обрабатывать другие сообщения, в том числе *WM_TIMER*. Вызов *EnableWindow(FALSE)* отключает кнопку Start на время вычислений. Без этой меры предосторожности возможен повторный вызов функции *OnBnClickedStart*. Повторный вызов функции *EnableWindow(TRUE)* включает кнопку Start, чтобы пользователь смог запустить таймер снова.

8. **Запрограммируйте функцию *OnTimer* в ComputeDlg.cpp.** При срабатывании таймера показание индикатора устанавливается в соответствии со значением *m_nCount*. Добавьте выделенный код:

```
void CComputeDlg::OnTimer(UINT nIDEvent)
{
   CProgressCtrl* pBar =
   (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
   pBar->SetPos(m_nCount * 100 / nMaxCount);

   CDialog::OnTimer(nIDEvent);
}
```

9. **Модифицируйте функцию *OnBnClickedCancel* в ComputeDlg.cpp.** При щелчке пользователем кнопки Cancel в процессе вычислений мы не уничтожаем диалоговое окно, а присваиваем *m_nCount* максимальное значение, в результате функция *OnBnClickedStart* закрывает диалоговое окно. Если вычисления не начаты, диалоговое окно можно закрыть напрямую. Добавьте выделенный код:

```
void CComputeDlg::OnBnClickedCancel()
{
   TRACE("entering CComputeDlg::OnBnClickedCancel\n");
   if (m_nCount == 0) {     // до нажатия кнопки Start
      CDialog::OnCancel();
   }
   else {              // идут вычисления
      m_nCount = nMaxCount; // принудительное завершение OnBnClickedStart
   }
}
```

10. **Отредактируйте класс *CEx11aView* в Ex11aView.cpp.** Измените виртуальную функцию *OnDraw*, как показано ниже, чтобы она выводила сообщение:

```
void CEx11aView::OnDraw(CDC* pDC)
{
   CEx11aDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDC->TextOut(0, 0, "Press the left mouse button here.");
}
```

Создайте функцию *OnLButtonDown* для обработки сообщения *WM_LBUTTONDOWN*. Выберите класс *CEx11aView* в Class View, в окне Properties щелкните кнопку Messages, выберите сообщение *WM_LBUTTONDOWN*, создайте функцию *OnLButtonDown* и добавьте в нее выделенный код:

```
void Cex11aView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CComputeDlg dlg;
   dlg.DoModal();

   CView::OnLButtonDown(nFlags, point);
}
```

Этот код открывает модальное диалоговое окно всякий раз, когда пользователь нажимает левую кнопку мыши, а курсор находится в окне представления.

Пока у вас еще открыт файл Ex11aView.cpp, введите оператор:

```
#include "ComputeDlg.h"
```

11. **Соберите и запустите приложение.** Поместив курсор в окно представления, щелкните левой кнопкой, чтобы активизировать диалоговое окно. Щелкните кнопку Start, затем — Cancel. Индикатор продвижения должен показывать состояние процесса вычислений.

Обработка в периоды простоя

До многопоточности разработчики программ для Windows использовали *периоды простоя* (idle time) для выполнения фоновых задач, например, разбивки документа на страницы. Теперь обработка во время простоя потеряла былое значение,

но ей по-прежнему находится применение. Каркас приложений вызывает виртуальную функцию-член *OnIdle* класса *CWinApp*, и вы можете переопределить ее для выполнения фоновых вычислений. *OnIdle* вызывается из цикла обработки сообщений MFC-библиотеки, который на самом деле сложнее, чем приведенная выше простая последовательность *GetMessage/TranslateMessage/DispatchMessage*.

Обычно по завершении своей работы функция *OnIdle* не вызывается до следующего опустошения очереди сообщений. Если вы переопределяете ее, вызывается ваш код, но при отсутствии непрерывного потока сообщений эта функция не вызывается. *OnIdle* в базовом классе обновляет кнопки панели инструментов и индикаторы состояния, а также «подчищает» указатели на временные объекты. Имеет смысл переопределять эту функцию для обновления состояния элементов пользовательского интерфейса. Ну, а то, что никакого кода не выполняется в отсутствие сообщений, не важно: пользовательский интерфейс и не должен в этом случае изменяться.

Примечание Переопределяя функцию *CWinApp::OnIdle*, не забудьте вызвать *OnIdle* базового класса. Иначе не произойдет ни обновления кнопок на панели инструментов, ни удаления временных объектов.

OnIdle вообще не вызывается, если пользователь работает в модальном диалоговом окне или выбирает что-то в меню. При необходимости фоновой обработки модальных диалоговых окон или меню придется написать обработчик сообщения *WM_ENTERIDLE*, но его надо добавить в класс *окна-рамки*, а не в класс «вид». Причина в том, что владельцем диалоговых окон всегда является основное окно-рамка приложения, а не окно представления. О взаимосвязи окна-рамки и окна представления см. главу 14.

Программирование многопоточных приложений

Как вы помните из главы 10, *процесс* (process) — это выполняемая программа, обладающая собственной памятью, описателями файлов и другими системными ресурсами. Процесс может содержать несколько параллельно исполняемых отрезков кода — *потоков* (thread). Но не ищите отдельного кода для разных потоков, потому что одну функцию могут вызывать несколько потоков. По большей части все пространство кода и данных процесса доступно всем его потокам. Два потока могут, например, обращаться к одним глобальным переменным. Потоками управляет ОС, и у каждого потока есть свой стек.

В Windows есть *потоки пользовательского интерфейса* (user-interface thread) и *рабочие* (worker thread). MFC-библиотека поддерживает оба вида. У потока пользовательского интерфейса есть окна, а значит, и свой цикл выборки сообщений, а у рабочего — нет.[1] Рабочие потоки легче программировать, и они обычно полезнее. Примеры в этой главе иллюстрируют рабочие потоки. Но в конце главы описывается приложение с применением потока пользовательского интерфейса.

[1] Такое деление весьма условно, так как у рабочего потока, если нужно, может быть цикл выборки сообщений. — *Прим. перев.*

Не забывайте, что даже в однопоточном приложении есть поток, называемый *основным* (main thread). В иерархии MFC-классов *CWinApp* является производным от *CWinThread*. В главе 2 было сказано, что *InitInstance* и *m_pMainWnd* — это элементы класса *CWinApp*. Вообще-то это неправда. Эти элементы объявлены в *CWinThread*, но, конечно же, наследуются классом *CWinApp*. Здесь важнее помнить, что приложение — это и *есть* поток.

Написание функции рабочего потока и запуск потока

Если вы еще не догадались сами, то знайте: для выполнения длительных вычислений рабочий поток эффективнее обработчика сообщений, содержащего вызов *PeekMessage*. Однако прежде чем думать о запуске рабочего потока, надо написать для него глобальную функцию. Она должна возвращать значение типа *UINT* и принимать в качестве параметра одно 32-разрядное значение, объявленное как *LPVOID*. При запуске потока через этот параметр можно передать ему все, что угодно. Поток выполняет свои вычисления и завершается, когда глобальная функция возвращает управление. Он завершается и при закрытии процесса, но лучше, чтобы рабочий поток завершался раньше — это поможет предотвратить утечку памяти.

Чтобы запустить поток (с функцией, скажем, *ComputeThreadProc*), программа делает вызов:

```
CWinThread* pThread = AfxBeginThread(ComputeThreadProc, GetSafeHwnd(),
THREAD_PRIORITY_NORMAL);
```

Код функции потока выглядит примерно так:

```
UINT ComputeThreadProc(LPVOID pParam)
{
   // процесс обработки
   return 0;
}
```

Функция *AfxBeginThread* возвращает управление сразу; она возвращает указатель на только что созданный объект «поток». Этот указатель позволяет приостановить и возобновить исполнение потока (*CWinThread::SuspendThread* и *ResumeThread*), но у объекта «поток» нет функции-члена для уничтожения потока. Второй параметр функции *AfxBeginThread* — 32-разрядное значение, передаваемое глобальной функции, а третий представляет собой код приоритета потока. После запуска рабочего потока оба потока исполняются независимо друг от друга. Windows распределяет время между ними (и потоками других процессов) согласно их приоритетам. Пока основной поток ожидает сообщение, ОС продолжает исполнение потока вычислений.

Общение основного потока с рабочим

Существует много способов связи между основным и рабочим потоками. Однако Windows-сообщение отправляться точно *не будет* — у рабочего потока нет цикла выборки сообщений. Простейшее средство коммуникации — глобальная переменная, поскольку все глобальные переменные доступны всем потокам процесса.

Допустим, рабочий поток в процессе вычислений увеличивает и проверяет значение глобальной целочисленной переменной, завершаясь, когда значение переменной достигает 100. Основной поток может принудительно завершить рабочий поток, присвоив глобальной переменной значение 100 или более. Следующий код, на первый взгляд, должен работать, и, если вы его протестируете, может, так и случится:

```
UINT ComputeThreadProc(LPVOID pParam)
{
   g_nCount = 0;
   while (g_nCount++ < 100) {
      // здесь выполняются какие-то вычисления
   }
   return 0;
}
```

Однако здесь есть одно «но», которое можно обнаружить, лишь посмотрев на сгенерированный ассемблерный код. Значение *g_nCount* загружается в регистр, увеличивается в нем же и переписывается обратно в *g_nCount*. Пусть *g_nCount* равно 40, и Windows прерывает рабочий поток сразу же после того, как он загружает это значение в регистр. Теперь управление получает основной поток и присваивает *g_nCount* значение 100. При возобновлении рабочий поток увеличивает значение регистра и записывает обратно в *g_nCount* число 41, стирая предыдущее значение 100. Результат — цикл потока не завершается!

Если же мы включим оптимизацию кода при компиляции, то получим дополнительную проблему. Переменную *g_nCount* компилятор размещает в регистре, причем значение переменной остается загруженным в него на протяжении всего цикла. Если основной поток изменит значение *g_nCount* в памяти, это не повлияет на цикл вычислений в рабочем потоке. (Однако, чтобы компилятор не хранил счетчик в регистре, можно объявить *g_nCount* как *volatile*.)

Но допустим, вы переписали процедуру потока:

```
UINT ComputeThreadProc(LPVOID pParam)
{
   g_nCount = 0;
   while (g_nCount < 100) {
      // здесь выполняются какие-то вычисления
      ::InterlockedIncrement((long*)&g_nCount);
   }
   return 0;
}
```

Функция *InterlockedIncrement* предотвращает обращение к переменной со стороны другого потока во время ее изменения. Теперь основной поток сможет завершить рабочий.

Итак, вы познакомились с некоторыми подводными камнями, подстерегающими программиста при использовании глобальных переменных. Иногда применение глобальных переменных оправданно, что иллюстрирует следующий пример, но есть и альтернативные методы, более гибкие; и мы еще обсудим их в этой главе.

Общение рабочего потока с основным

Вполне разумно использовать глобальную переменную, значение которой периодически проверяет рабочий поток, но что, если то же самое сделает основной поток? Помните «жадную» функцию? Безусловно, нежелательно, чтобы основной поток входил в цикл — ведь это лишь потеря процессорного времени, да и ваша программа перестанет обрабатывать сообщения. Для связи рабочего потока с основным лучше передавать сообщения Windows, так как у основного потока уже есть цикл выборки сообщений. Однако это подразумевает, что у основного потока есть окно (видимое или нет), а у рабочего потока — его описатель.

А как рабочий поток получит описатель? Для этого служит 32-разрядный параметр уже рассмотренной функции потока. Вы просто передаете описатель в вызове *AfxBeginThread*. Но почему бы вместо него не передать указатель на объект «окно» С++? Это может оказаться опасным, так как нельзя полагаться на то, что этот объект будет существовать постоянно, и, кроме того, разным потокам не разрешается совместно использовать объекты MFC-классов. (Это правило не распространяется на объекты классов, производных непосредственно от *CObject*, и на объекты таких простых классов, как *CRect* или *CString*.)

Как посылать сообщение: *синхронно* (send) или *асинхронно* (post)? Асинхронная передача предпочтительнее, так как синхронная может вызвать повторное вхождение в MFC-код для выборки сообщений основного потока, а это чревато проблемами. А какое сообщение следует посылать? Да любое пользовательское.

Пример Ex11b

Внешне Ex11b выглядит абсолютно так же, как программа Ex11a. Однако, обратившись к ее коду, вы заметите различия. Вычисления выполняются в рабочем, а не в основном потоке. Значение счетчика хранится в глобальной переменной *g_nCount*, которой присваивается максимальное значение в обработчике кнопки Cancel диалогового окна. Завершаясь, поток посылает сообщение диалоговому окну, что вызывает завершение функции *DoModal*.

Классы документа, представления, рамки и приложения остались теми же за исключением имен; не изменился и диалоговый ресурс. Класс модального диалогового окна по-прежнему называется *CComputeDlg*, но его реализация изменилась. Конструктор, обработчик таймера и функции обмена данными практически те же.

Следующий код показывает определение глобальной переменной и глобальную функцию потока из файла \Ex11b\ComputeDlg.cpp с компакт-диска. Заметьте: возврат из функции (и завершение потока) происходит, когда *g_nCount* превышает определенное максимальное значение. Однако, прежде чем завершиться, функция асинхронно отправляет диалоговому окну пользовательское сообщение:

```
int g_nCount = 0;

UINT ComputeThreadProc(LPVOID pParam)
{
   volatile int nTemp; // volatile, иначе компилятор перестарается с оптимизацией

   for (g_nCount = 0; g_nCount < CComputeDlg::nMaxCount;
```

```
                  ::InterlockedIncrement((long*) &g_nCount)) {
      for (nTemp = 0; nTemp < 50000; nTemp++) {
         // просто занимаем процессор
      }
   }
   // WM_THREADFINISHED - это пользовательское сообщение
   ::PostMessage((HWND) pParam, WM_THREADFINISHED, 0, 0);
   g_nCount = 0;
   return 0; // завершаем поток
}
```

Показанный ниже обработчик *OnBnClickedStart* связан с кнопкой Start диалогового окна. Его задача — запустить таймер и рабочий поток. Третий параметр *AfxBeginThread* позволяет изменять приоритет рабочего потока — например, вычисления замедляются, если установить самый низкий приоритет (*THREAD_PRIORITY_LOWEST*):

```
void CComputeDlg::OnBnClickedStart()
{
   m_nTimer = SetTimer(1, 100, NULL); // 1/10 секунды
   ASSERT(m_nTimer != 0);
   GetDlgItem(IDC_START)->EnableWindow(FALSE);
   AfxBeginThread(ComputeThreadProc, GetSafeHwnd(),
                  THREAD_PRIORITY_NORMAL);
}
```

Обработчик *OnBnClickedCancel* (см. ниже) связан с кнопкой Cancel диалогового окна. Он присваивает переменной *g_nCount* максимальное значение, что приводит к завершению рабочего потока:

```
void CComputeDlg::OnBnClickedCancel()
{
   if (g_nCount == 0) { // до нажатия кнопки Start
      CDialog::OnCancel();
   }
   else { // идут вычисления
      g_nCount = nMaxCount; // принудительное завершение потока
   }
}
```

Обработчик *OnThreadFinished* сопоставляется отправляемому в диалоговое окно пользовательскому сообщению *WM_THREADFINISHED*. Он заставляет функцию *DoModal* завершить свою работу:

```
LRESULT CComputeDlg::OnThreadFinished(WPARAM wParam, LPARAM lParam)
{
   GetDlgItem(IDC_START)->EnableWindow(TRUE);

   CDialog::OnOK();
   return 0;
}
```

Синхронизация потоков с использованием событий

Применение глобальной переменной — грубое, но эффективное средство связи потоков. Попробуем что-нибудь более совершенное и попытаемся мыслить в терминах *синхронизации* потоков, а не простой передачи информации. Наши потоки должны тщательно синхронизировать свое взаимодействие.

Событие (event) — один из типов объектов ядра (процессы и потоки тоже относятся к объектам ядра), предоставляемых Windows для синхронизации потоков. В пределах конкретного процесса событие определяется уникальным 32-разрядным описателем и для совместного использования несколькими процессами может идентифицироваться по имени (или описатель события может дублироваться в другом процессе). Объект «событие» находится либо в *свободном* (TRUE, signaled state), либо в *занятом* (FALSE, unsignaled state) состоянии. События бывают с *ручным* (manual reset) и *автоматическим сбросом* (autoreset). Мы рассмотрим события последние, так как они идеально подходят для синхронизации двух процессов.

Вернемся к примеру с рабочим потоком. Мы хотим, чтобы основной поток (поток пользовательского интерфейса) сигнализировал рабочему потоку, когда начинать и прекращать работу. Поэтому нам понадобятся события «*запустить*» (start) и «*уничтожить*» (kill). MFC предоставляет для этого удобный класс *CEvent*, производный от *CSyncObject*. Конструктор по умолчанию создает объект Win32 «событие» с автосбросом в занятом состоянии. Если определить события как глобальные объекты, любой поток сможет легко обратиться к ним. Когда основному потоку надо запустить или уничтожить рабочий поток, он переводит соответствующее событие в свободное состояние вызовом *CEvent::SetEvent*.

Теперь рабочий поток должен наблюдать за состоянием двух событий и должным образом реагировать, когда одно из них переходит в свободное состояние. Для этого в MFC есть класс *CSingleLock*, но проще вызвать Win32-функцию *WaitForSingleObject*. Она приостанавливает поток, пока заданный объект не освободится. В приостановленном состоянии поток не занимает процессорного времени, и это хорошо. Первый параметр функции *WaitForSingleObject* — описатель события. В качестве значения этого параметра можно использовать сам объект *CEvent*, потому что он наследует от класса *CSyncObject* оператор *HANDLE*, который возвращает описатель события, хранящийся в открытой переменной-члене. Второй параметр определяет период тайм-аута. Если его значение *INFINITE*, освобождения объекта «событие» функция ожидает неопределенно долго (вечно), а когда оно равно 0, функция сразу возвращает управление с результатом *WAIT_OBJECT_0*, если событие было в свободном состоянии.

Пример Ex11c

В этой программе для синхронизации рабочего и основного потоков используются два события. Большая часть кода Ex11c взята из примера Ex11b, но класс *CComputeDlg* реализован иначе. Файл StdAfx.h содержит строку, необходимую для класса *CEvent*:

```
#include <afxmt.h>
```

В программе два глобальных объекта-события, как показано ниже. Заметьте: конструкторы создают события Windows до начала исполнения главной процедуры:

```
CEvent g_eventStart; // создает события с автосбросом
CEvent g_eventKill;
```

Сначала лучше всего изучить глобальную функцию рабочего потока. Как и в Ex11b, она увеличивает значение счетчика *g_nCount*. Рабочий поток запускает функция *OnInitDialog*, а не обработчик сообщений от кнопки Start. При первом вызове функция *WaitForSingleObject* ждет события «запустить», которое переводится в свободное состояние обработчиком кнопки Start. Параметр *INFINITE* означает, что поток будет ждать столько, сколько надо. Второй вызов *WaitForSingleObject* отличается от первого — здесь задержка нулевая. Этот вызов расположен в основном цикле вычислений и просто проверяет, не освобождено ли событие «kill» обработчиком кнопки Cancel. Если это так, поток завершается.

```
UINT ComputeThreadProc(LPVOID pParam)
{
    volatile int nTemp;

    ::WaitForSingleObject(g_eventStart, INFINITE);
    TRACE("starting computation\n");
    for (g_nCount = 0; g_nCount < CComputeDlg::nMaxCount;
                     g_nCount++) {
        for (nTemp = 0; nTemp < 10000; nTemp++) {
            // имитируем вычисления
        }
        if (::WaitForSingleObject(g_eventKill, 0) == WAIT_OBJECT_0) {
            break;
        }
    }
    // Сообщить окну-владельцу об окончании работы
    ::PostMessage((HWND) pParam, WM_THREADFINISHED, 0, 0);
    g_nCount = 0;
    return 0; // завершаем поток
}
```

А вот функция *OnInitDialog*, вызываемая при инициализации диалогового окна (заметьте: она запускает рабочий поток, который бездействует, пока не освободится событие «запустить»):

```
BOOL CComputeDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    AfxBeginThread(ComputeThreadProc, GetSafeHwnd());
    return TRUE; // возвращать TRUE, если не установлен фокус на элементе управления
                 // ИСКЛЮЧЕНИЕ: Страницы свойств OCX должны возвращать FALSE
}
```

Далее обработчик кнопки Start устанавливает событие «запустить» в свободное состояние и тем самым запускает цикл вычислений рабочего потока:

```
void CComputeDlg::OnBnClickedStart()
{
    m_nTimer = SetTimer(1, 100, NULL); // 1/10 секунды
    ASSERT(m_nTimer != 0);
    GetDlgItem(IDC_START)->EnableWindow(FALSE);
    g_eventStart.SetEvent();
}
```

Обработчик кнопки Cancel устанавливает событие «уничтожить» в свободное состояние, что вызывает завершение цикла вычислений рабочего потока:

```
void CComputeDlg::OnBnClickedCancel()
{
    if (g_nCount == 0) { // до нажатия кнопки Start
        // До уничтожения потока его надо запустить
        g_eventStart.SetEvent();
    }
    g_eventKill.SetEvent();
}
```

Обратите внимание на весьма неуклюжее применение события «запустить», когда пользователь отменяет окно, не запустив процесса вычислений. Аккуратнее было бы определить новое событие «отменить» и заменить в функции *ComputeThreadProc* первый вызов *WaitForSingleObject* на вызов *WaitForMultipleObjects*. Если бы *WaitForMultipleObjects* обнаруживала событие «отменить», это могло бы приводить к немедленному завершению потока.

Блокировка потоков

Пример блокировки потока — первый вызов *WaitForSingleObject* в функции *ComputeThreadProc*. Поток просто прекращает выполнение до освобождения события. Способов заблокировать поток много. Можно, например, вызвать Win32-функцию *Sleep*, чтобы «усыпить» поток на 500 мс. Блокировку потока вызывают и функции, которые обращаются к устройствам вроде коммуникационных портов или дисков. Во времена Win16 эти функции захватывали процессор до завершения своей работы, а в Win32 они позволяют выполняться другим процессам и потокам.

Избегайте блокирующих вызовов в основном потоке пользовательского интерфейса. Если заблокировать основной поток, он не сможет обрабатывать сообщения, и работа программы покажется замедленной. Если у вас есть задача, требующая интенсивных операций дискового ввода/вывода, поместите соответствующий код в рабочий поток и синхронизируйте его с основным потоком.

Будьте осторожны с вызовами, способными заблокировать рабочий поток на неограниченное время. Сверьтесь с документацией, можно ли для данной конкретной операции ввода/вывода установить интервал задержки. Вызов может блокировать поток навсегда, однако последний все равно завершится при завершении основного потока процесса, но тогда не исключены утечки памяти. Можно также вызвать из основного потока функцию Win32 *TerminateThread*, но и тогда проблемы утечки памяти не избежать.

Критические секции

Помните сложности с доступом к глобальной переменной *g_nCount*? Если требуется организовать совместный доступ нескольких потоков к глобальным данным и для этого вам нужна большая гибкость, чем та, что предоставляют простые операторы типа *InterlockedIncrement*, лучше всего подойдут *критические секции* (critical sections). События хороши для «сигнализации», а критические секции (секции кода, требующие монопольного доступа к совместно используемым данным) — для управления доступом к данным.

MFC предоставляет класс *CCriticalSection* — «обертку» описателя критической секции Windows. Его конструктор вызывает функцию *InitializeCriticalSection*, функции-члены *Lock* и *Unlock* вызывают функции *EnterCriticalSection* и *LeaveCriticalSection* соответственно, а деструктор вызывает *DeleteCriticalSection*. Вот как можно использовать этот класс для защиты глобальных данных:

```
CCriticalSection g_cs; // Глобальные переменные, доступные из всех потоков
int g_nCount;
void func()
{
    g_cs.Lock();
    g_nCount++;
    g_cs.Unlock();
}
```

Допустим, ваша программа отслеживает показания времени как часы, минуты и секунды, а каждое из этих значений хранится в отдельной целочисленной переменной. Теперь представим, что значения времени совместно используются двумя потоками. Поток *A* изменяет значение времени и прерывается потоком *B* после обновления часов, но до обновления минут и секунд. Результат: поток *B* получает недостоверные показания времени.

Если вы создаете для данного формата времени класс C++, то сможете легко управлять доступом к данным, сделав элементы данных закрытыми и предусмотрев открытые функции-члены. Таков показанный в следующем примере класс *CHMS*. Заметьте: в нем есть переменная-член типа *CCriticalSection*. Так что с каждым объектом *CHMS* связан объект «критическая секция».

Заметьте: другие функции-члены вызывают функции-члены *Lock* и *Unlock*. Если поток *A* исполняется в середине *SetTime*, поток *B* будет блокирован вызовом *EnterCriticalSection* в *GetTotalSecs* до того, как поток *A* не вызовет *LeaveCriticalSection*. Функция *IncrementSecs* вызывает *SetTime*, что означает наличие вложенных критических секций. Это допустимо, так как Windows отслеживает уровни их вложенности.

Класс *CHMS* отлично работает, если вы применяете его для конструирования глобальных объектов. Если же потоки вашей программы совместно используют указатели на объекты в куче, вы столкнетесь с рядом других проблем. Каждый поток должен определять, не удален ли объект другим потоком, а значит, нужна синхронизация доступа к указателям.

HMS.H

```
[illegible]
```

Мы не приводим пример программы, использующей класс *CHMS*, но на компакт-диске в подкаталоге Ex11c есть файл HMS.h. Создавая многопоточную программу, вы сможете применять этот класс для совместного использования глобальных объектов между потоками. При этом вам не понадобятся другие функции, связанные с синхронизацией потоков.

Мьютексы и семафоры

Как мы уже говорили, подробные сведения об объектах синхронизации вы найдете в книге Джеффри Рихтера. *Мьютекс* (mutex — сокращение от mutual exclu-

sion — взаимное исключение) или семафор могут потребоваться, если нужно управлять доступом к данным, совместно используемым разными процессами, так как критическая секция доступна лишь в пределах одного процесса. Мьютексы и семафоры совместно используются процессами и идентифицируются по именам[1].

Потоки пользовательского интерфейса

MFC-библиотека обеспечивает хорошую поддержку потоков пользовательского интерфейса. Вы создаете класс, производный от *CWinThread*, и вызываете переопределенную версию *AfxBeginThread* для запуска потока. У этого производного класса есть своя функция *InitInstance* и, что самое важное, свой цикл выборки сообщений — это позволяет конструировать связанные с ним окна и обработчики сообщений.

Зачем нужен поток пользовательского интерфейса? Если вы хотите работать с несколькими окнами верхнего уровня, их можно создать и управлять ими из основного потока. Но допустим, вы разрешаете пользователю запускать несколько экземпляров вашего приложения и хотите, чтобы все они совместно обращались к общей памяти. Можно сделать так, чтобы в одном процессе исполнялось несколько потоков пользовательского интерфейса, а пользователи думали, что выполняются отдельные процессы. Именно так работает Windows Explorer. Запустите утилиту SPYXX и убедитесь сами.

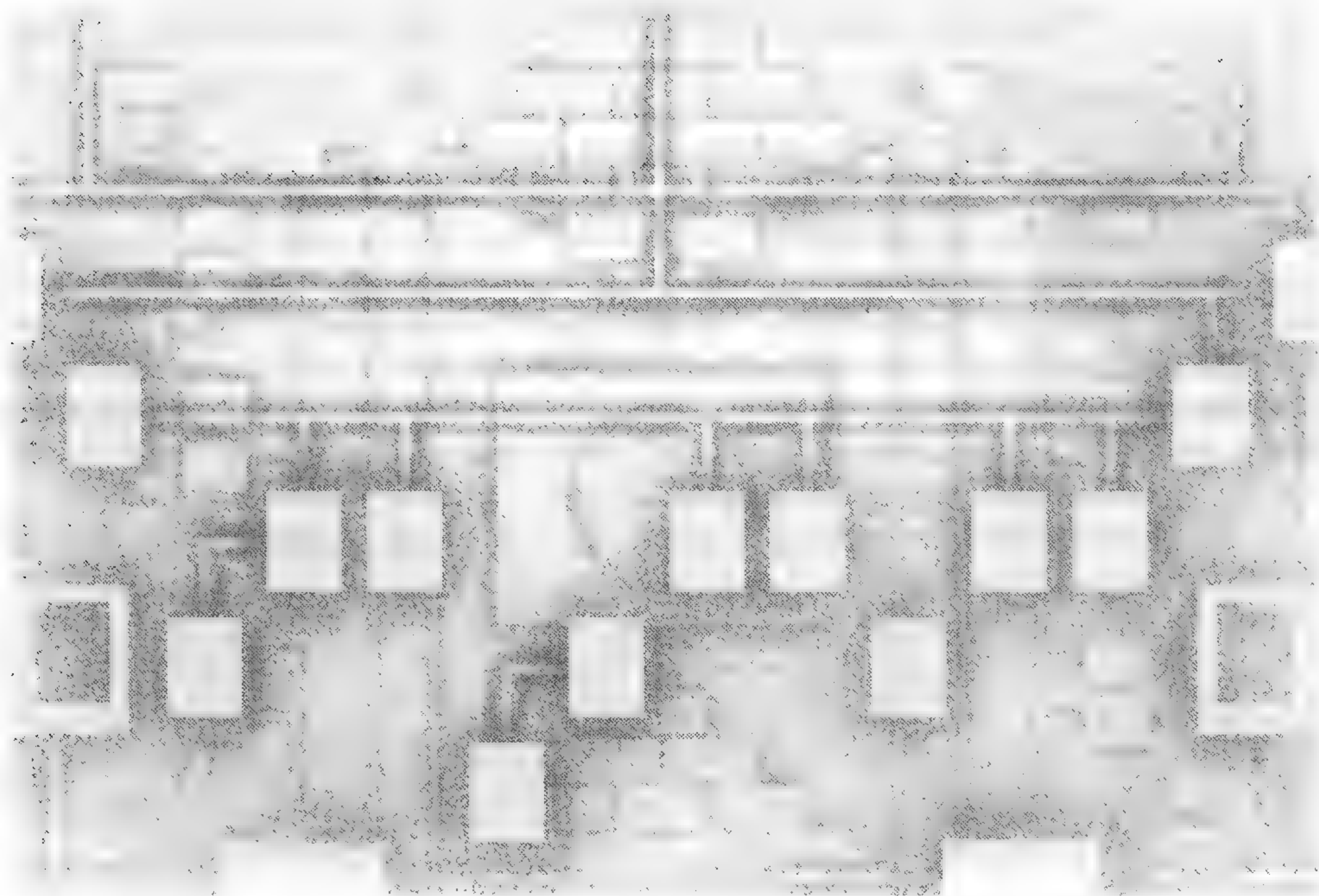
При запуске второго и последующих потоков без уловок не обойтись, так как пользователь фактически каждый раз запускает новый процесс. При запуске второго процесса тот сигнализирует первому запустить второй поток и завершается. Второй процесс обнаруживает первый либо с помощью Win32-функции *FindWindow*, либо путем объявления общей секции данных. Общие секции данных детально рассматриваются в книге Джеффри Рихтера.

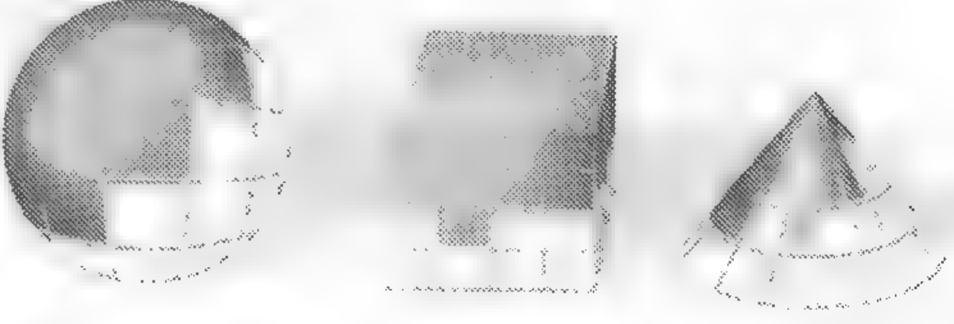
---

[1] А также путем дублирования описателей аналогично событиям. — *Прим. перев.*

ЧАСТЬ 3

АРХИТЕКТУРА «ДОКУМЕНТ-ВИД» В MFC





ГЛАВА

12

Меню, быстрые клавиши, поля ввода с форматированием и окна свойств

В предыдущих примерах большинство действий в программах выполнялось по щелчку кнопки мыши. Даже когда удобнее было выбрать из меню, мы все равно щелкали кнопки, потому что сообщения мыши в окне представления каркаса MFC обрабатываются просто и эффективно. Если вы хотите, чтобы выполнение действий в программе начиналось после выбора команд меню, придется освоить другие элементы каркаса приложений.

Эта глава посвящена меню и *архитектуре маршрутизации команд* (command routing architecture). Попутно вы познакомитесь с понятиями *рамка* (frame) и *документ*, уясните взаимосвязь между этими элементами каркаса приложений и уже известным вам элементом *вид*. Вы научитесь использовать редактор меню для визуального создания меню и мастерами, доступными в окне Class View, для связи функций-членов классов «документ» и «вид» с командами меню. Кроме того, вы освоите специальные функции-члены, позволяющие обновлять командный пользовательский интерфейс (в том числе изменять состояние команд в меню) и научитесь программировать *быстрые клавиши*, ускоряющие доступ к командам меню.

Вы, верно, уже устали от окружностей и диалоговых окон, поэтому предлагаем познакомиться с другими строительными блоками MFC — элементом управления *поле ввода с форматированием* (rich edit control), который предоставляет богатые возможности редактирования текстов, и *окнами свойств* (property sheet), идеально подходящими для задания и редактирования значений свойств.

Классы основного окна-рамки и документа

До сих пор мы использовали окно представления так, словно это единственное окно приложения. Но в SDI-приложении (Single Document Interface) это окно располагается внутри другого окна — основного окна-рамки приложения. Именно ему принадлежат заголовок и меню. Клиентскую область основного окна занимают всевозможные дочерние окна, в том числе окно представления, окно панели инструментов и окно строки состояния (рис. 12-1). Каркас приложений управляет взаимодействием между окном-рамкой и окном представления, доставляя сообщения от первого второму.

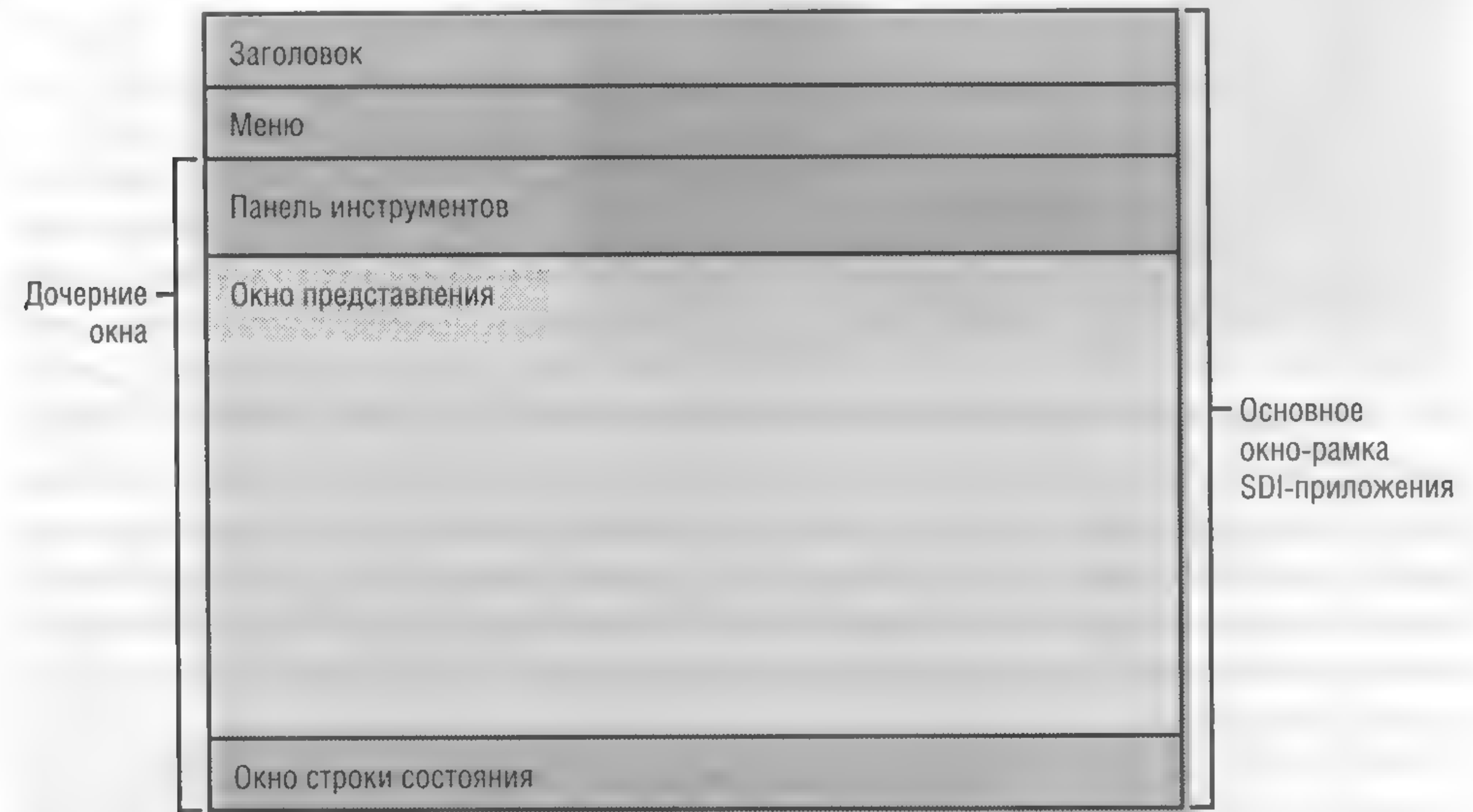


Рис. 12-1. *Дочерние окна в основном окне-рамке SDI-приложения*

Посмотрите еще раз на файлы какого-нибудь проекта, которые мы сгенерировали, используя MFC Application Wizard. Файлы MainFrm.h и MainFrm.cpp содержат код класса основного окна-рамки приложения, производного от класса *CFrameWnd*. В других файлах (например, Ex12aDoc.h и Ex12aDoc.cpp) хранится код класса документа, производного от *CDocument*. С этой главы мы будем работать с MFC-классом «документ». Начнем с того, что каждый объект «вид» связан только с одним объектом «документ» и функция-член *GetDocument* возвращает указатель на этот документ. В главе 15 мы продолжим изучение взаимодействия «документ-вид».

Меню Windows

Меню Microsoft Windows — знакомый всем компонент приложения, состоящий из горизонтального списка элементов верхнего уровня; с ним связаны меню, открывающиеся при выборе пользователем какого-либо из его элементов. Обычно для окна-рамки определяется ресурс меню по умолчанию, загружаемый при создании этого окна. Можно определить и ресурс меню, не связанный с каким-либо окном-рамкой. В этом случае ваша программа должна вызывать функции, необходимые для загрузки и активизации меню.

Ресурс меню полностью определяет начальное состояние меню: отключение определенных команд, отметка галочкой или группировка с помощью разделителей. Можно создавать многоуровневые выпадающие меню. Если команда меню первого уровня связана с подменю, то рядом с этой командой появляется стрелка, указывающая вправо, как у команды Windows в меню Debug на рис. 12-2.

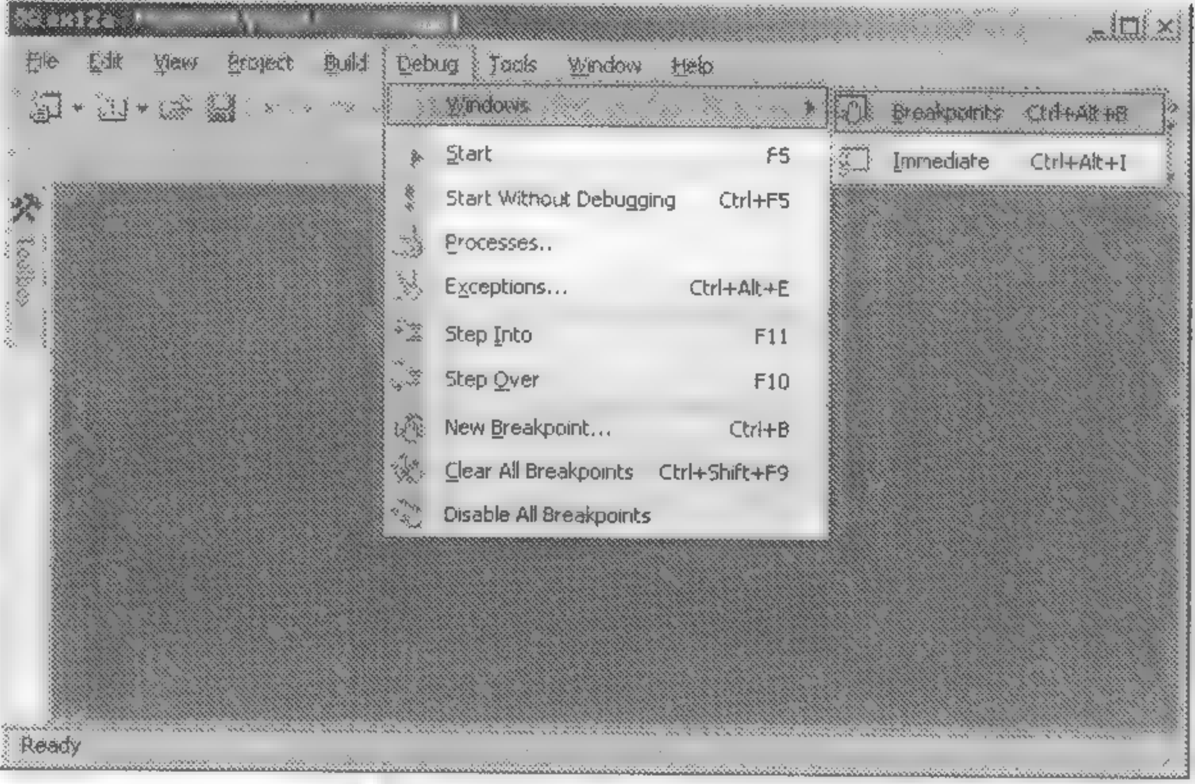


Рис. 12-2. *Многоуровневые раскрывающиеся меню (в Microsoft Visual C++ .NET)*

В Visual C++ в .NET включен простой в применении редактор меню, позволяющий создавать и редактировать меню в режиме WYSIWYG. Для каждой команды меню есть окно свойств, где задаются все характеристики этой команды. Полученное определение ресурса сохраняется в RC-файле проекта. Каждой команде меню присваивается свой идентификатор, например *ID_FILE_OPEN*, определяемый в файле Resource.h.

MFC-библиотека расширяет функциональность стандартных меню Windows. Каждый элемент меню может дополняться строкой подсказки, отображающейся в строке состояния окна-рамки при выборе этого элемента. По сути подсказки — это строковые ресурсы Windows, связанные с командами меню с применением обычных идентификаторов. Для редактора меню и вашей программы эти подсказки — всего лишь часть определения команды в меню.

Быстрые клавиши

В большинстве команд меню есть подчеркнутый символ. В среде разработки Visual C++ .NET (и многих других программах) нажатие сочетания клавиш Alt+F, а затем S активизирует команду Save из меню File. Такая система *быстрых клавиш* — стандартный для Windows способ ускорения доступа к командам меню с клавиатуры. Если вы посмотрите на файл описания ресурсов приложения (или заглянете в диалоговое окно свойств для команды в редакторе меню), то заметите, что символам, подчеркнутым в командах меню, предшествует знак амерперсанд (&).

В Windows предусмотрен еще один способ связывания последовательностей клавиш с командами меню. Ресурс клавиш-акселераторов — это таблица, описывающая комбинации клавиш и соответствующие им идентификаторы команд. Через запись в такой таблице команду Copy из меню Edit (с идентификатором *ID_EDIT_COPY*) можно связать, скажем, с комбинацией клавиш Ctrl+C. Быстрая клавиша необя-

зательно должна соответствовать какой-либо команде в меню. Даже если в меню Edit нет команды Copy, ничто не мешает вам назначить комбинацию клавиш Ctrl+C для инициирования команды *ID_EDIT_COPY*.

Примечание Если быстрая клавиша связана с командой меню или кнопкой панели инструментов, то при отключении команды или кнопки отключается и эта клавиша.

Обработка команд

Как вы уже видели в главе 2, в каркасе приложений есть весьма изощренная система маршрутизации командных сообщений, поступающих при выборе элементов меню, нажатии быстрых клавиш, а также при щелчке кнопок на панелях инструментов или в диалоговых окнах. Командные сообщения можно также отправлять, вызвав функцию *CWnd::SendMessage* или *PostMessage*. Сообщения идентифицируют *#define*-константы, часто назначаемые в редакторе ресурсов. У каркаса приложений собственный набор идентификаторов внутренних командных сообщений, например, *ID_FILE_PRINT* или *ID_FILE_OPEN*. А файл Resource.h вашего проекта содержит уникальные идентификаторы конкретного приложения.

Большинство командных сообщений генерируется в окне-рамке приложения, и именно здесь, не будь каркаса приложений, вам пришлось бы размещать обработчики команд. Система же маршрутизации команд позволяет обрабатывать эти сообщения практически где угодно. Обнаружив командное сообщение от окна-рамки, каркас приложений начинает поиск соответствующих обработчиков в таком порядке.

В SDI-приложении	В MDI-приложении
Класс «вид»	Класс «вид»
Класс «документ»	Класс «документ»
Класс основного окна-рамки SDI	Класс дочернего окна-рамки MDI
Класс «приложение»	Класс основного окна-рамки MDI

В большинстве приложений обработчик конкретной команды присутствует в одном классе, но если в программе с одним окном представления два обработчика — и в классе «вид», и в классе «документ», то, поскольку окно представления в иерархии распределения команд занимает более высокую ступень, вызывается обработчик класса «вид».

Как создать функцию-обработчик команды? Требования к определению такого обработчика аналогичны уже известным вам правилам определения обработчика оконных сообщений. Для этого нужна сама функция, соответствующий элемент в карте сообщений и прототип функции. Допустим, в меню есть команда Zoom (с идентификатором *IDM_ZOOM*), которую надо обрабатывать в классе «вид». В этом случае добавьте сначала в файл реализации класса «вид» такой код:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   ON_COMMAND(IDM_ZOOM, OnZoom)
END_MESSAGE_MAP()
```

```
void CMyView::OnZoom()
{
   // код обработки командного сообщения
}
```

Затем введите в файл заголовка класса *CMyView* прототип (перед макросом *DECLARE_MESSAGE_MAP*):

```
afx_msg void OnZoom();
```

Конечно, Visual Studio .NET автоматизирует процесс создания обработчиков командных сообщений точно так же, как и обработчиков оконных сообщений. Но как именно это работает, вы узнаете, познакомившись с примером Ex12a.

Обработка командных сообщений в производных классах

Система распределения команд — лишь одна сторона обработки командных сообщений. Не менее важна иерархия классов. Заглянув в исходный код MFC-классов, вы увидите в карте сообщений множество элементов *ON_COMMAND*. При создании производного класса от одного из этих базовых классов, например от *CView*, производный класс наследует все функции карты сообщений *CView*, включая функции-обработчики командных сообщений. Чтобы переопределить какую-то функцию карты сообщений базового класса, в производный класс надо добавить как функцию, так и соответствующую запись карты сообщений.

Обновление командного пользовательского интерфейса

Весьма часто приходится менять внешний вид элементов меню, чтобы отразить внутреннее состояние программы. Так, если в меню Edit имеется команда Clear All (Очистить все), то ее нужно отключать, если очищать нечего. В меню Windows-программ вы, конечно, видели отключенные и помеченные галочками команды.

При программировании в Win32 синхронизировать состояние элементов меню в соответствии с состоянием приложения не так просто. Каждый участок кода, изменяющий внутреннее состояние программы, должен содержать операторы, обновляющие меню. В MFC-библиотеке реализован другой подход, основанный на вызове специальной функции-обработчика, которая и обновляет командный пользовательский интерфейс при каждом открытии меню. Аргумент этой функции — объект *CCmdUI* — содержит указатель на соответствующий элемент меню. Используя этот указатель, функция-обработчик может изменить внешний вид команды меню. Подобные обработчики применимы к элементам раскрывающихся меню, но не к меню верхнего уровня, которые постоянно присутствуют на экране. Такой обработчик нельзя использовать, например, для отключения меню File.

Требования к определению обработчиков, обновляющих командный пользовательский интерфейс, сходны с требованиями к определению обработчиков команд. Вам нужна собственно функция, специальный элемент карты сообщений и, конечно, прототип функции. Идентификатор (у нас это *IDM_ZOOM*) применяется тот же, что и для команды. Вот что надо добавить к файлу кода для класса «вид»:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   ON_UPDATE_COMMMAND_UI(IDM_ZOOM, OnUpdateZoom)
```

```
END_MESSAGE_MAP()

void CMyView::OnUpdateZoom(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(m_bZoomed); // m_bZoomed - переменная-член класса
}
```

Прототип следует включить в заголовочный файл класса (перед макросом *DECLARE_MESSAGE_MAP*):

```
afx_msg void OnUpdateZoom(CCmdUI* pCmdUI);
```

Понятно, что мастера Class View автоматизируют и процесс вставки обработчиков, обновляющих командный пользовательский интерфейс.

Команды, генерируемые диалоговыми окнами

Допустим, вы хотите, чтобы одна из кнопок диалогового окна посылала командное сообщение. Идентификаторы команд должны попадать в диапазон 0x8000—0xDFFF, который редактор ресурсов использует для элементов меню. Если присвоить кнопке диалогового окна идентификатор из этого диапазона, она будет генерировать команду, маршрутизируемую каркасом приложений. Сначала каркас пересылает эту команду основному окну-рамке, так как именно оно — владелец всех диалоговых окон. Затем доставка команды пойдет обычным путем; если в классе «вид» есть обработчик командного сообщения от кнопки, он и будет ее обрабатывать. Чтобы значение идентификатора не вышло за указанный выше диапазон, используйте диалоговое окно Resource Symbols в редакторе ресурсов, позволяющее определить идентификатор до назначения его кнопке.

Встроенные меню каркаса приложений

Создавать меню для каждого окна-рамки с нуля нет нужды — в библиотеке MFC уже определено несколько полезных меню (рис. 12-3) со всеми функциями-обработчиками команд.

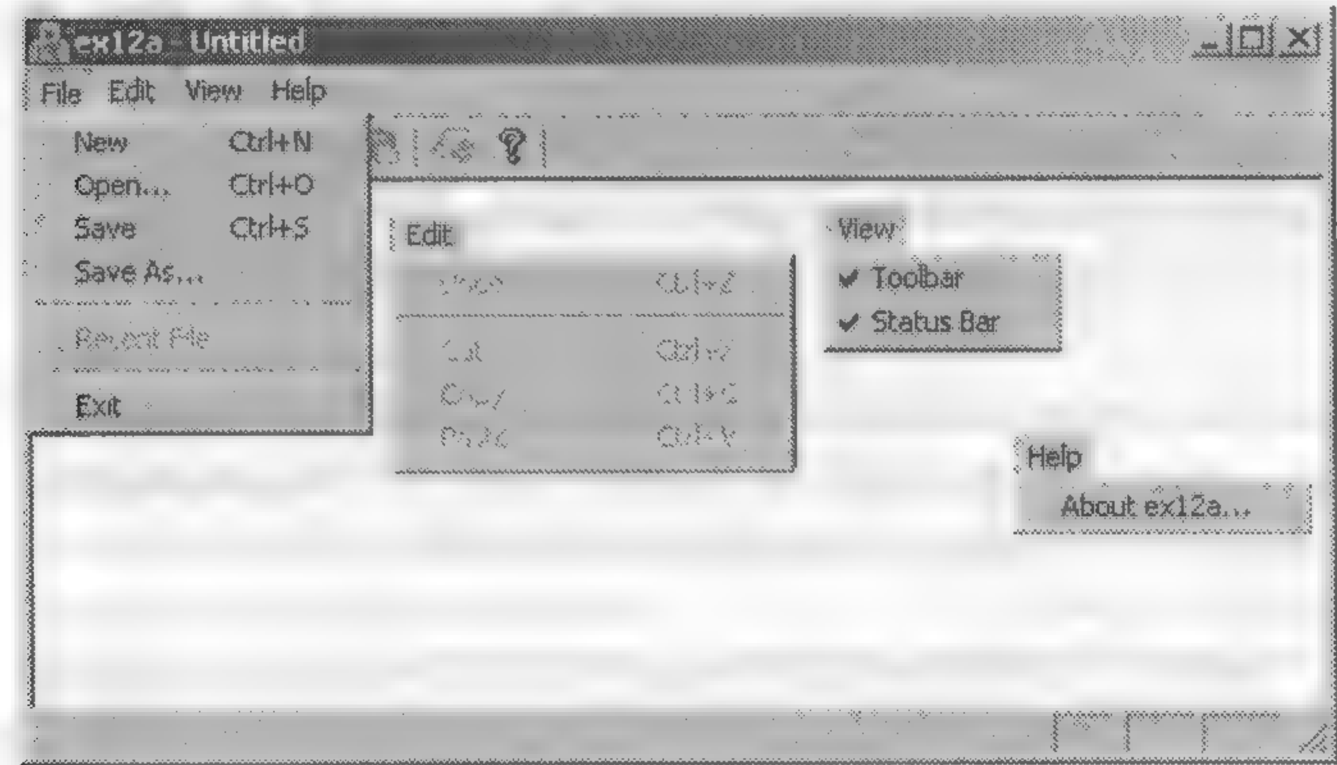


Рис. 12-3. *Стандартные меню в окне-рамке SDI-приложения*

Состав меню и обработчиков командных сообщений зависит от параметров, определенных в MFC Application Wizard. Так, если сбросить флажок Printing And Print Preview, не будет элементов меню Print и Print Preview. Так как поддержка печати необязательна, соответствующие элементы в карте сообщений в классе *CView* не определены, а генерируются для конкретного производного класса. Вот почему элементы вроде приведенных ниже определены в классе *CMyView*, а не в *CView*:

```
ON_COMMAND(ID_FILE_PRINT,  CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW,  CView::OnFilePrintPreview)
```

Включение и отключение команд в меню

Каркас приложений способен сам отключить команду меню, не найдя нужный обработчик на текущем маршруте команды, и это избавляет от необходимости писать обработчики *ON_UPDATE_COMMAND_UI*. Подобное поведение можно запретить, присвоив значение FALSE переменной *m_bAutoMenuEnable* класса *CFrameWnd*.

Допустим, у вас есть два представления одного документа, но только в первом классе «вид» имеется обработчик команды *IDM_ZOOM*. В этом случае команда Zoom в меню окна-рамки будет активна, только когда активно первое окно представления. Это же верно в отношении стандартных меню Edit, Cut, Copy и Paste, создаваемых каркасом элементов. Они отключаются, если в производном классе «вид» или «документ» не определены обработчики этих командных сообщений.

Редактирование текста в MFC

В Windows два средства редактирования текста: обычное поле ввода (edit control) и поле ввода с форматированием (rich edit common control). Их можно задействовать как элементы управления в диалоговых окнах, но можно сделать и так, чтобы они выглядели, как окна представления. Эту гибкость обеспечивают MFC-классы *CEditView* и *CRichEditView*.

Класс *CEditView*

В основе этого класса — элемент управления Windows «поле ввода». MFC Application Wizard позволяет задать *CEditView* в качестве базового для вашего класса «вид». При работе с объектом *CEditView* доступны все функции-члены как *CView*, так и *CEdit*. Множественное наследование здесь не применяется — есть просто несколько «фокусов», в том числе создание оконных подклассов. Класс *CEditView* реализует функции, с которыми связываются команды работы с буфером обмена «вырезать», «копировать» и «вставить», так что эти команды разрешены в меню Edit. По умолчанию объем текста ограничен — не более 1 048 575 символов, хотя вы вправе изменить его, отправив в элемент управления сообщение *EM_LIMITTEXT*. Однако это ограничение дополнительно зависит от ОС и типа элемента управления (с одной или многими строками). Подробнее см. библиотеку MSDN.

Класс *CRichEditView*

Этот класс окна представления базируется на элементе управления «поле ввода с форматированием» и потому поддерживает большие объемы текста и форматирование. Класс *CRichEditView* предназначен для совместного использования с классами *CRichEditDoc* и *CRichEditCntrItem*, что позволяет реализовать полноценное контейнерное приложение ActiveX.

Класс *CRichEditCtrl*

Этот класс — оболочка для элемента управления «поле ввода с форматированием», и его обычно применяют для создания простого текстового редактора. Этим мы и займемся в примере Ex12a: возьмем обычный класс «вид», производный от *CView*, и «закроем» всю клиентскую область окна большим полем ввода с форматированием, которое автоматически меняет свои размеры при масштабировании окна. Класс *CRichEditCtrl* содержит десятки полезных функций-членов, в том числе из базового класса *CWnd*. В этой главе мы задействуем такие функции (табл. 12-1):

Табл. 12-1. Популярные функции класса *CRichEditCtrl*

Функция	Описание
Create	Создает окно элемента управления «поле ввода с форматированием» (вызывается из обработчика *WM_CREATE* родительского окна).
SetWindowPos	Задает размер и положение окна ввода (устанавливает размер окна, так чтобы оно «закрывало» клиентскую области окна представления).
GetWindowText	Получает из элемента управления неформатированный текст (текст в формате RTF возвращают другие функции).
SetWindowText	Записывает в элемент управления неформатированный текст.
GetModify	Возвращает флаг, значение которого равно *TRUE*, если текст изменен [текст изменяется, когда пользователь что-то набирает в элементе управления или когда программа вызывает *SetModify(TRUE)*]. В противном случае — *FALSE*.
SetModify	Устанавливает признак модификации *TRUE* или *FALSE*.
GetSel	Возвращает флаг, значение которого указывает, выделил ли пользователь какой-либо текст.
SetDefaultCharFormat	Устанавливает характеристики форматирования по умолчанию.
SetSelectionCharFormat	Устанавливает характеристики форматирования выделенного текста.

Примечание Если вы добавили поле ввода с форматированием в диалоговое окно, то в функции *InitInstance* класса приложения надо вызвать функцию *AfxInitRichEdit*.

Пример Ex12a

Здесь иллюстрируется доставка команд от меню и быстрых клавиш документу и окну представления. Класс «вид» приложения — производный от *CView* и содержит элемент управления «поле ввода с форматированием». Команды нового раскрывающегося меню Transfer, направляемые окну представления, пересылают данные между объектами «вид» и «документ», а команда меню Clear Document удаляет содержимое документа. В меню Transfer команда Store Data In Document отключена, если в окне представления ничего нет. Команда Clear Document, расположенная в меню Edit, становится неактивной, когда документ пуст. На рис. 12-4 представлен первый вариант программы Ex12a в действии.

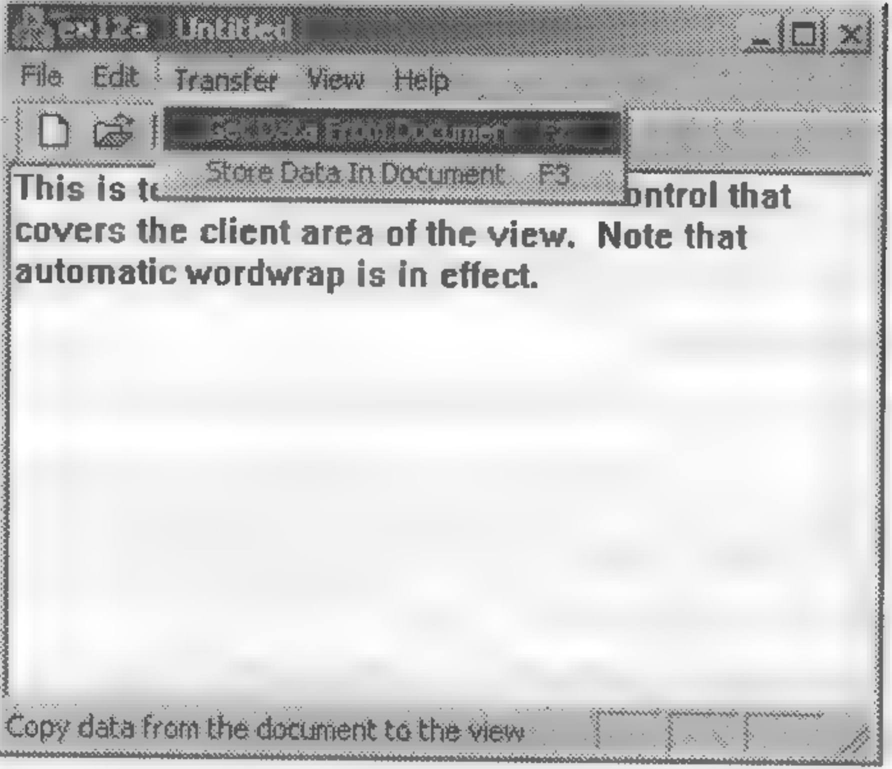
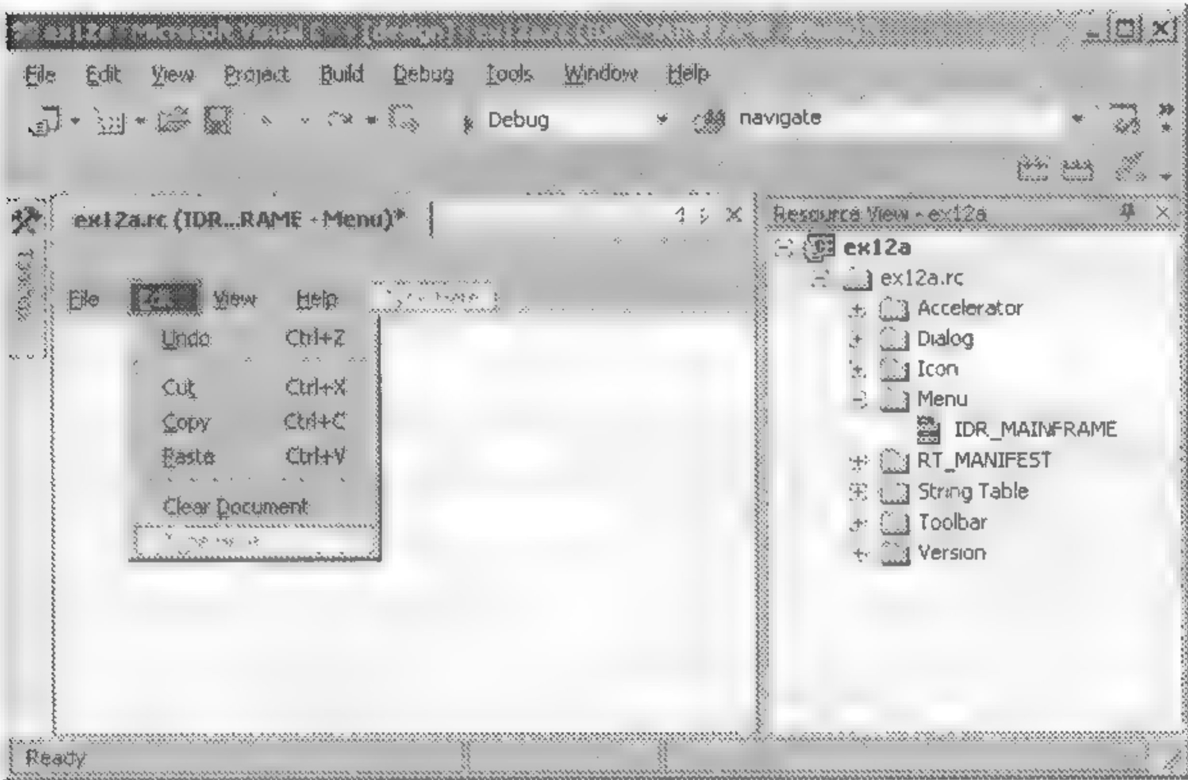


Рис. 12-4. *Программа Ex12a в действии*

Если полностью задействовать архитектуру «документ-вид», можно заставить поле ввода с форматированием хранить свой текст в документе, но сделать это трудно. Вместо этого определим в документе переменную-член *m_strText* класса *CString*, содержимое которой пользователь может перемещать в элемент управления и из него. Начальное значение *m_strText* — «Hello»; выбор команды Clear Document из меню Edit очищает эту строку. Поработав с примером, вы лучше поймете, чем отличается документ от своего представления.

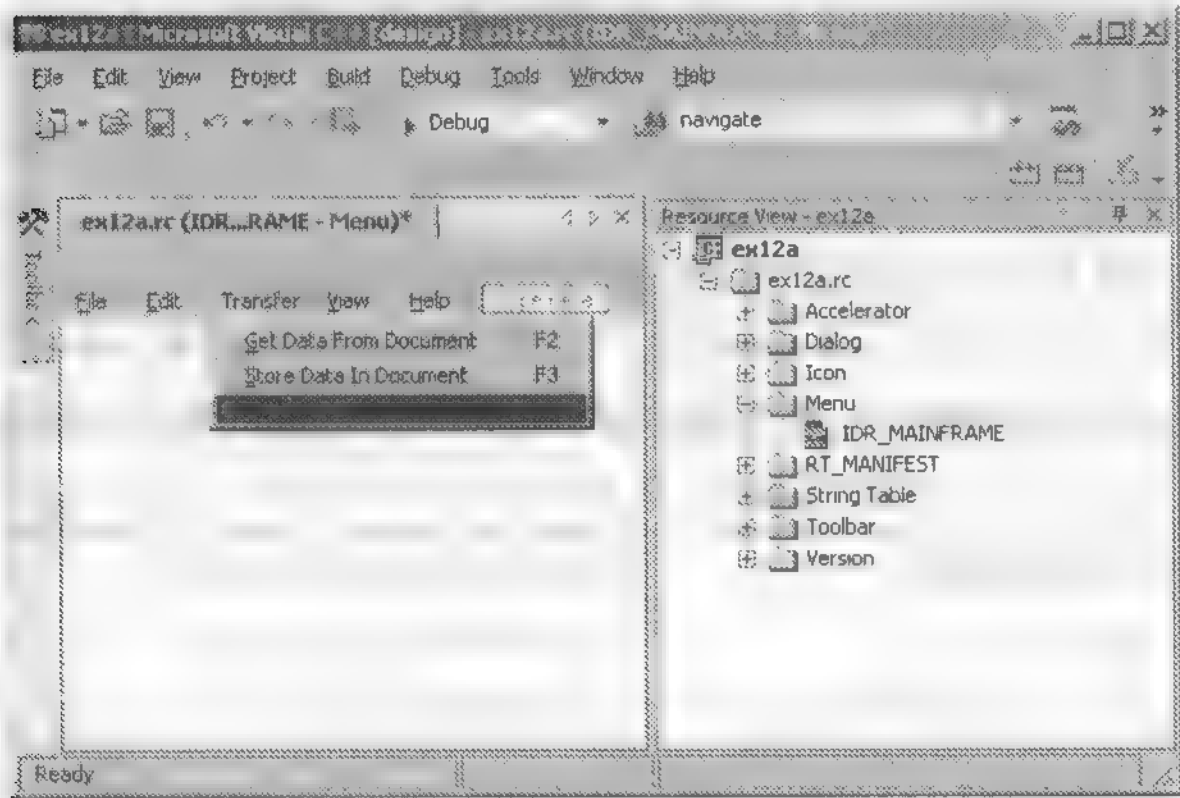
Первая часть примера Ex12a позволяет поупражняться в работе с WYSIWYG-редактором меню и редактором быстрых клавиш с применением мастеров окна Class View. Собственно программирования на C++ здесь будет совсем немного.

1. **Используя MFC Application Wizard, создайте проект Ex12a.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC Application, а имени проекта — Ex12a. На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **Отредактируйте основное меню приложения в редакторе ресурсов.** В окне Resource View отредактируйте ресурс меню *IDR_MAINFRAME*, чтобы добавить разделитель и команду Clear Document в меню Edit:



Совет Редактор меню интуитивно понятен, но в первый раз вам, наверное, будет неясно, как вставить команду в середину меню. Просто щелкните правой кнопкой место, куда нужно вставить команду, и в контекстном меню выберите Insert New. Чтобы добавить разделитель в контекстном меню, выберите Insert Separator.

Теперь добавьте меню Transfer и определите его элементы:



MFC назначил новым элементам меню следующие идентификаторы в диалоговом окне Resource Symbols. (Заметьте: \t — это символ табуляции, но введите именно **\t**, не нажимайте клавишу Tab.)

Меню	Команда	Идентификатор команды
Edit	Clear &Document	*ID_EDIT_CLEARDOCUMENT*
Transfer	&Get Data From Document\tF2	*ID_TRANSFER_GETDATAFROM-DOCUMENT*
Transfer	&Store Data In Document\tF3	*ID_TRANSFER_STOREDATAIN-DOCUMENT*

Добавив команды в меню, последовательно щелкните каждый элемент правой кнопкой и в контекстном меню выберите Properties и введите строки подсказки

в диалоговом окне Properties. Эти подсказки появляются в строке состояния программы при выделении команды меню.

3. **Используя редактор ресурсов, укажите быстрые клавиши.** Откройте таблицу быстрых клавиш *IDR_MAINFRAME*, дважды щелкнув ее значок в окне Resource View. Затем щелкните пустую строку в конце таблицы и добавьте такие записи:

Идентификатор быстрой клавиши	Клавиша
ID_TRANSFER_GETDATAFROMDOCUMENT	VK_F2
ID_TRANSFER_STOREDATAINDOCUMENT	VK_F3

Не забудьте установить в False свойства Ctrl, Alt и Shift, чтобы отключить модификаторы Ctrl, Alt и Shift.

4. **В окне Properties утилиты Class View создайте в классе *CEx12aView* обработчики командных сообщений и сообщений обновления командного пользовательского интерфейса.** Выберите класс *CEx12aView* и добавьте функции-члены:

Идентификатор объекта	Сообщение	Имя функции-члена
ID_TRANSFER_ GETDATAFROMDOCUMENT	*COMMAND*	*OnTransferGetdatafromdocument*
ID_TRANSFER_ STOREDATAINDOCUMENT	*COMMAND*	*OnTransferStoredataindocument*
ID_TRANSFER_ STOREDATAINDOCUMENT	*UPDATE_COMMAND_UI*	*OnUpdateTransfer- Storedataindocument*

5. **В окне Properties утилиты Class View создайте в классе документа обработчики командных сообщений и сообщений обновления командного пользовательского интерфейса.** Выберите класс *CEx12aDoc* и добавьте такие функции-члены.

Идентификатор объекта	Сообщение	Имя функции-члена
ID_EDIT_CLEARDOCUMENT	*COMMAND*	*OnEditClearDocument*
ID_EDIT_CLEARDOCUMENT	*UPDATE_COMMAND_UI*	*OnUpdateEditCleardocument*

6. **Добавьте файл Ex12aDoc.cpp строку:**

```
#include "Ex12aView.h"
```

7. **Добавьте переменную-член типа *CString* в класс *CEx12aDoc*.** Отредактируйте файл Ex12aDoc.h или в окне Class View добавьте код:

```
public:
    CString m_strText;
```

8. **Отредактируйте функции-члены класса «документ» в файле Ex12aDoc.cpp.** Функция *OnNewDocument* сгенерирована Visual Studio .NET. Каркас приложений вызывает ее, когда документ создается впервые и когда пользователь выбирает в меню File команду New. Наша версия этой функции присваивает текст строковой переменной.

```
BOOL CEx12aDoc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;
   m_strText = "Hello (from CEx12aDoc::OnNewDocument)";
   return TRUE;
}
```

Обработчик сообщения Edit Clear Document очищает *m_strText*, а обработчик обновления пользовательского интерфейса отключает (делает блеклой) соответствующую команду меню, если строка уже пуста. Помните: каркас приложений вызывает *OnUpdateEditCleardocument*, когда открывается меню Edit.

```
void CEx12aDoc::OnEditClearDocument()
{
   m_strText.Empty();
   // Отобразить изменения в представлениях
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CEx12aView* pView = (CEx12aView*) GetNextView(pos);
      pView->m_rich.SetWindowText(m_strText);
   }
}

void CEx12aDoc::OnUpdateEditCleardocument(CCmdUI *pCmdUI)
{
   pCmdUI->Enable(!m_strText.IsEmpty());
}
```

9. **Добавьте переменную типа *CRichEditCtrl* в класс *CEx12aView*.** Отредактируйте файл Ex12aView.h вручную или используя Class View, добавив строку:

```
public:
   CRichEditCtrl m_rich;
```

10. **В окне Properties утилиты Class View создайте в классе *CEx12aView* обработчики сообщений *WM_CREATE* и *WM_SIZE*.** Функция *OnCreate* создает «поле ввода с форматированием». Здесь размер этого элемента управления равен 0, так как мы еще не знаем размера окна представления. Вот код обоих обработчиков:

```
int CEx12aView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   CRect rect(0, 0, 0, 0);
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;
   m_rich.Create(ES_AUTOVSCROLL | ES_MULTILINE | ES_WANTRETURN |
      WS_CHILD | WS_VISIBLE | WS_VSCROLL, rect, this, 1);
   return 0;
}
```

Windows посылает сообщение *WM_SIZE* окну представления, как только определяется начальный размер окна, и далее — при каждом изменении размера рамки пользователем. Наш обработчик корректирует размер поля ввода с форматированием так, чтобы оно заполняло всю клиентскую область окна представления:

```
void CEx12aView::OnSize(UINT nType, int cx, int cy)
{
    CRect rect;
    CView::OnSize(nType, cx, cy);
    GetClientRect(rect);
    m_rich.SetWindowPos(&wndTop, 0, 0, rect.right - rect.left,
                        rect.bottom - rect.top, SWP_SHOWWINDOW);
}
```

11. **Отредактируйте обработчики команд меню в файле Ex12aView.cpp.** Заготовки этих функций созданы Visual Studio .NET при создании обработчиков команд на шаге 4. Функция *OnTransferGetdatafromdocument* копирует текст из переменной-члена класса «документ» и помещает его в поле ввода с форматированием. Затем функция сбрасывает флаг модификации элемента управления. Обработчика, обновляющего командный пользовательский интерфейс, здесь нет.

```
void CEx12aView:: OnTransferGetdatafromdocument ()
{
    CEx12aDoc* pDoc = GetDocument();
    m_rich.SetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);
}
```

Функция *OnTransferStoredataindocument* копирует текст из поля ввода с форматированием в строку документа и сбрасывает флаг модификации элемента управления. Соответствующий обработчик, обновляющий командный пользовательский интерфейс, отключает команду меню, если содержимое элемента управления не менялось с момента последнего обмена содержимым с документом.

```
void CEx12aView::OnTransferStoredataindocument()
{
    CEx12aDoc* pDoc = GetDocument();
    m_rich.GetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);
}

void CEx12aView::OnUpdateTransferStoredataindocument(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_rich.GetModify());
}
```

12. **Соберите и протестируйте приложение Ex12a.** После запуска приложения команда Clear Document в меню Edit должна быть доступной. Выберите из меню Transfer команду Get Data From Document, и в окне появится некий текст. От-

редактируйте его и выберите команду Store Data In Document. Теперь эта команда должна стать недоступной. Выберите команду Clear Document и снова вызовите команду Get Data From Document.

Окна свойств

Вы уже видели их в Visual C++ .NET и других современных программах для Windows. Этот удобный элемент пользовательского интерфейса позволяет разместить в маленьком диалоговом окне большой объем информации, да еще разбитой на категории. Пользователь выбирает страницы (вкладки), щелкая их ярлычки. В Windows предусмотрен элемент управления *набор вкладок* (tab control), который можно вставить в диалоговое окно, но вам скорее всего понадобится обратное — разместить диалоговые окна в наборе вкладок. В MFC-библиотеке есть соответствующая поддержка — *окно свойств* (property sheet). Кстати, отдельные его диалоговые окна — вкладки — называются *страницами свойств* (property page).

Создание окна свойств

Вот как создать окно свойств с помощью инструментов Visual C++ .NET.

1. Создайте в редакторе ресурсов набор шаблонов диалоговых окон примерно одинакового размера. Их заголовки — строки, отображаемые на ярлычках.
2. Используя MFC Class Wizard, сгенерируйте класс для каждого шаблона. В качестве базового класса выберите *CPropertyPage*. Добавьте переменные-члены для элементов управления.
3. С помощью MFC Class Wizard создайте класс, производный от *CPropertySheet*.
4. Добавьте в класс окна свойств по одной переменной для каждого класса страниц свойств.
5. В конструкторе класса окна свойств вызовите функцию-член *AddPage* для каждой страницы, указывая адрес внедряемого объекта страницы свойств.
6. Создайте в своем приложении объект класса, производного от *CPropertySheet*, и вызовите *DoModal*. В вызове конструктора укажите заголовок окна (впоследствии его можно будет изменить обращением к функции *CPropertySheet::SetTitle*).
7. Запрограммируйте обработку кнопки Apply.

Обмен данными в окне свойств

Каркас приложений размещает в окне свойств три кнопки (рис. 12-5). Учтите: каркас вызывает DDX-код для страницы свойств всякий раз, когда пользователь переключается на нее или с нее на другую страницу. Кроме того, как и следовало ожидать, каркас приложений вызывает DDX-код для страницы, когда пользователь щелкает кнопку OK, обновляя тем самым элементы данных страницы. Теперь понятно, что все элементы данных всех страниц свойств обновляются после щелчка кнопки OK, и все это без программирования на C++!

Примечание В обычном модальном диалоговом окне при щелчке кнопки Cancel все изменения теряются, и переменные-члены класса диалогового окна остаются неизменными. Однако в окне свойств переменные-члены об-

новляются, даже если пользователь модифицирует содержимое одной страницы, переходит к другой, а потом отменяет изменения в окне свойств щелчком кнопки Cancel.

Что делает кнопка Apply? Да ничего, если только вы не напишете для нее какого-нибудь кода. Она даже не будет активна. Чтобы активизировать ее для конкретной страницы, проверьте, внесены ли туда изменения, и, если да, установите флаг модификации страницы, вызвав *SetModified(TRUE)*.

Если вы активизировали кнопку Apply, для нее можно написать функцию-обработчик в вашем классе страницы свойств, переопределив виртуальную функцию *CPropertyPage::OnApply*. Не пытайтесь разобраться с обработкой сообщений страницами свойств, руководствуясь в качестве аналогии поведением обычных модальных диалогов, — это «две большие разницы». При щелчке любой кнопки каркас приложений получает сообщение *WM_NOTIFY* и, если нажата кнопка OK или Apply, вызывает DDX-код для страницы, а затем *для всех страниц* вызывает виртуальную функцию *OnApply* и сбрасывает флаг модификации, отключая тем самым кнопку Apply. Не забудьте, что к DDX-коду уже обращались для обновления переменных-членов на всех страницах свойств и поэтому переопределять *OnApply* надо лишь в одном классе страницы свойств.

Что включать в функцию *OnApply* — ваше дело; один из вариантов — отправить пользовательское сообщение объекту, создавшему окно свойств. Обработчик сообщения может считывать значения переменных-членов для страниц свойств и выполнять соответствующие операции, пока окно свойств остается на экране.

И снова пример Ex12a

Теперь мы добавим в Ex12a окно свойств, в котором пользователь сможет изменять характеристики шрифта в поле ввода с форматированием. Конечно, можно было бы взять стандартный диалог *CFontDialog*, но как же вы тогда научитесь создавать окна свойств? На рис. 12-5 показано, что получится, если вы продолжите работу над Ex12a.

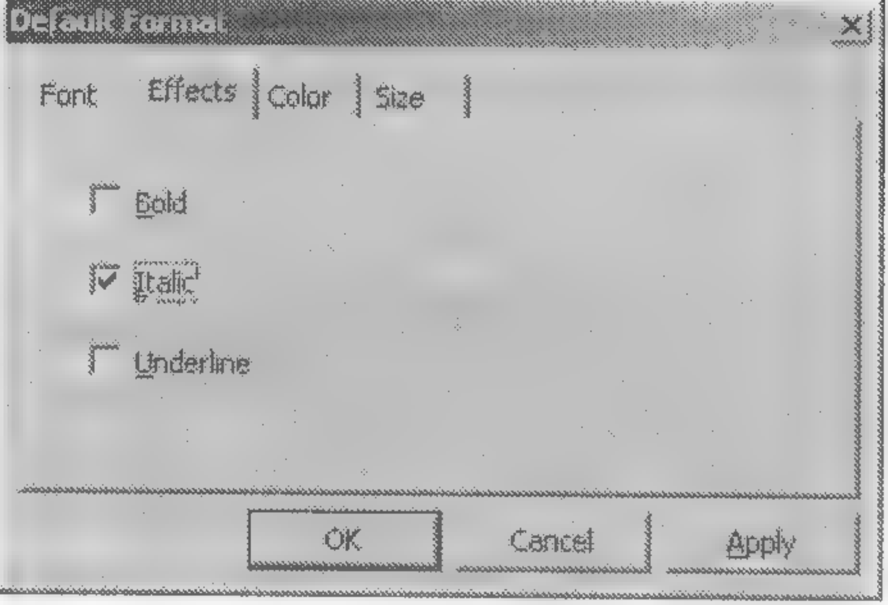
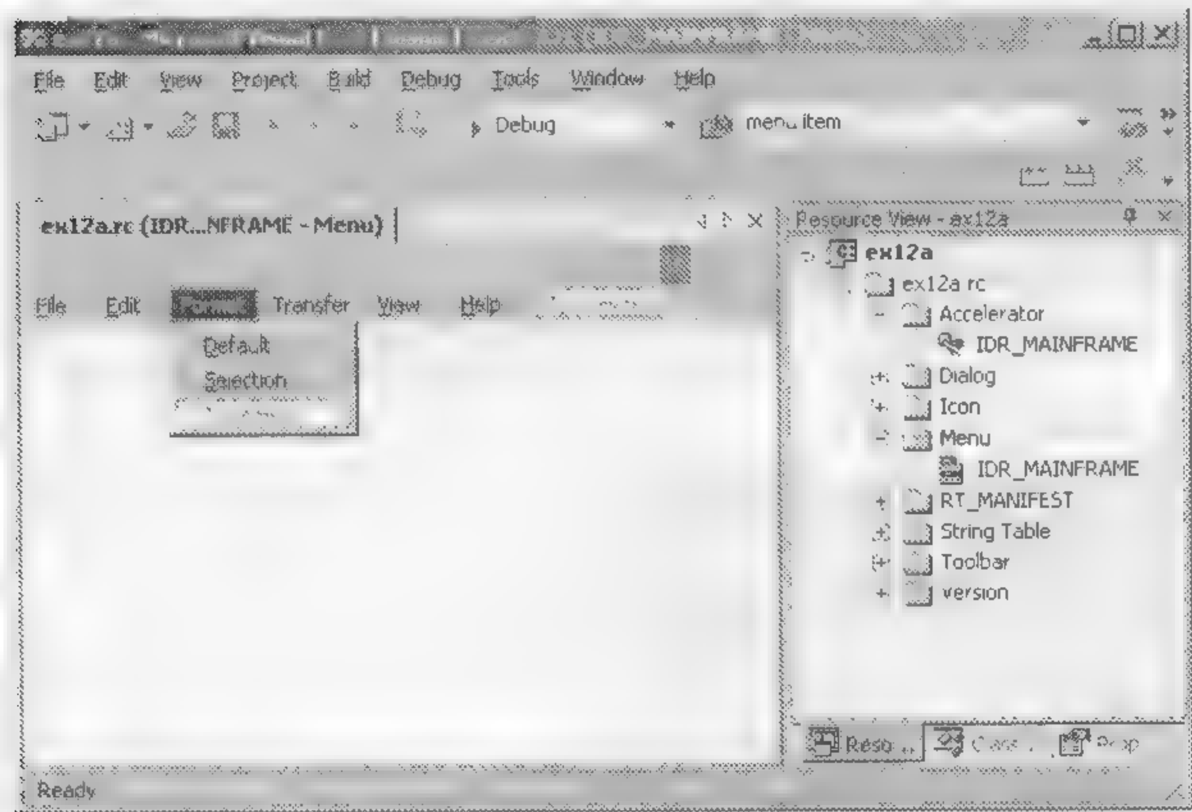


Рис. 12-5. *Окно свойств в программе Ex12a*

Соберите приложение Ex12a (если еще не собрали) в соответствии с изложенными ранее инструкциями. Ну, а если вы уже поработали с Ex12a, сделайте так.

1. **Используя редактор ресурсов, измените основное меню приложения.** В окне Resource View отредактируйте ресурс меню *IDR_MAINFRAME*, добавив к нему меню Format:



MFC присвоило следующие идентификаторы элементам нового меню:

Элемент	Идентификатор команды
&Default	*ID_FORMAT_DEFAULT*
&Selection	*ID_FORMAT_SELECTION*

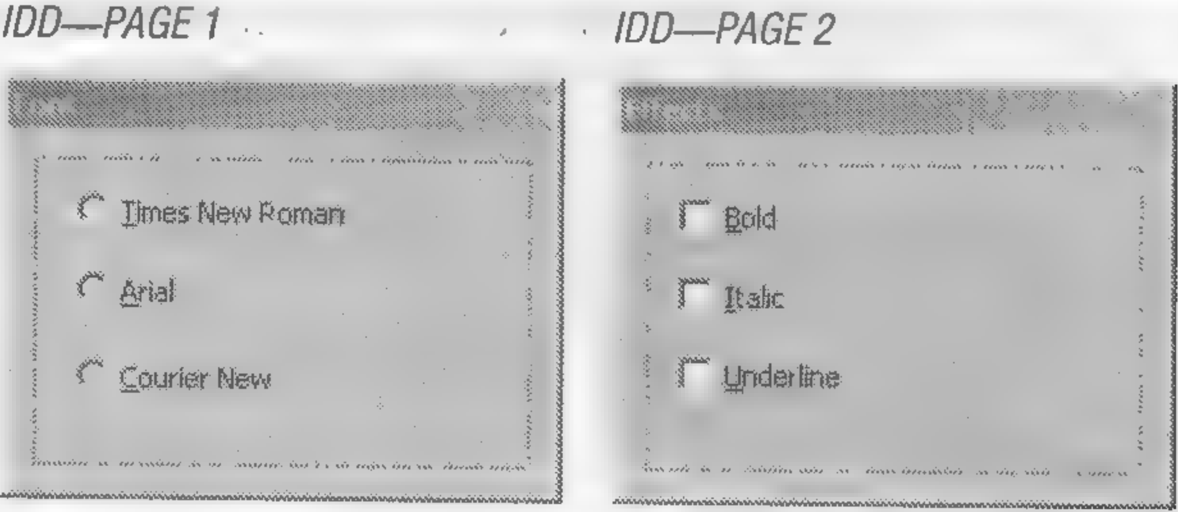
Задайте для этих пунктов меню соответствующие строки подсказки.

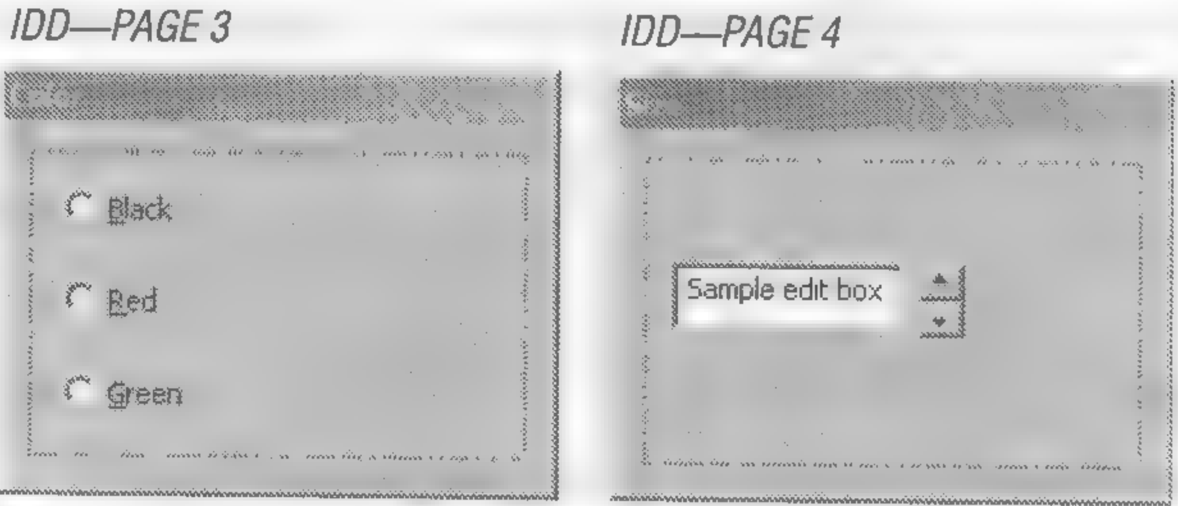
2. **В окне Properties утилиты Class View создайте в классе «вид» обработчики командных сообщений и сообщений обновления командного пользовательского интерфейса.** В окне Class View выберите класс *CEx12aView* и добавьте функции-члены:

Идентификатор объекта	Сообщение	Имя функции-члена
ID_FORMAT_DEFAULT	*COMMAND*	*OnFormatDefault*
ID_FORMAT_SELECTION	*COMMAND*	*OnFormatSelection*
ID_FORMAT_SELECTION	*UPDATE_COMMAND_UI*	*OnUpdateFormatSelection*

3. **В графическом редакторе создайте четыре шаблона страниц свойств.** Щелкните правой кнопкой RC-файл в Resource View и в контекстном меню выберите Add Resource. В диалоговом окне Add Resource выберите шаблон небольшой страницы свойств (small property page). Шаблоны и соответствующие идентификаторы показаны на рисунке (см. на след. стр.).

   Назначьте элементам управления в диалоговых окнах (страницах свойств) идентификаторы из приведенной ниже таблицы. У элемента *«наборный счетчик»* (spin control) установите в TRUE свойства Auto Buddy и Set Buddy Integer. У *переключателей* (radio button) *IDC_FONT* и *IDC_COLOR* установите в TRUE свойство Group. Минимальное значение элемента управления *IDC_FONTSIZE* установите равным 8, максимальное — 24.





Затем, используя MFC Class Wizard, создайте классы *CPage1, CPage2, CPage3* и *CPage4* — все производные от *CPropertyPage*. Разместите эти классы в файлах Property.h и Property.cpp, каждый раз изменяя имена файлов в полях ввода. Согласитесь на предложение Visual Studio .NET объединить файлы, щелкнув кнопку Yes. Добавьте следующие переменные-члены.

Диалоговое окно	**Элемент управления**	**Идентификатор**	**Тип**	**Переменная-член**
IDD_PAGE1	Первый переключатель	*IDC_FONT*	int	*m_nFont*
IDD_PAGE2	Флажок Bold	*IDC_BOLD*	BOOL	*m_bBold*
IDD_PAGE2	Флажок Italic	*IDC_ITALIC*	BOOL	*m_bItalic*
IDD_PAGE2	Флажок Underline	*IDC_UNDERLINE*	BOOL	*m_bUnderline*
IDD_PAGE3	Первый переключатель	*IDC_COLOR*	int	*m_nColor*
IDD_PAGE4	Поле ввода	*IDC_FONTSIZE*	int	*m_nFontSize*
IDD_PAGE4	Наборный счетчик	*IDC_SPIN1*		

Наконец, в окне Properties окна Class View переопределите функцию-обработчик *OnInitDialog* в *CPage4*.

4. **Используя MFC Class Wizard, сгенерируйте класс *CFontSheet*, производный от *CPropertySheet*.** Код класса сгенерируйте в файлах Property.h и Property.cpp, в которых уже находится код классов страниц свойств. Содержимое этих файлов показано ниже (добавляемый код выделен).

Property.h

```
[illegible]
    virtual BOOL OnApply();
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
[illegible]
```

см. след. стр.



```
virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);
```

```
}

[illegible]
[illegible]

[illegible]
{
   [illegible]
public:
   CPage1 m_page1;
   CPage2 m_page2;
   CPage3 m_page3;
   CPage4 m_page4;
[illegible]
   [illegible]
      [illegible]
   [illegible]
      [illegible]
   [illegible]

[illegible]
   [illegible]
};
```

Property.cpp

```
[illegible]

[illegible]
[illegible]
[illegible]

CView* g_pView;
[illegible]
[illegible]

[illegible]

[illegible]
{
   m_nFont = -1;
}

[illegible]
{


BOOL CPage1::OnApply()
{
```

см. след. стр.

```
    TRACE("CPage1::OnApply\n");
    g_pView->SendMessage(WM_USERAPPLY);
    return TRUE;
}

BOOL CPage1::OnCommand(WPARAM wParam, LPARAM lParam)
{
    SetModified(TRUE);
    return CPropertyPage::OnCommand(wParam, lParam);
}
[illegible]
{
    TRACE("Entering CPage1::DoDataExchange - %d\n",
          pDX->m_bSaveAndValidate);
    [illegible]
    [illegible]
}

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
{
    [illegible]
    [illegible]
    [illegible]
}

[illegible]
}
}

BOOL CPage2::OnCommand(WPARAM wParam, LPARAM lParam)
{
    SetModified(TRUE);
    return CPropertyPage::OnCommand(wParam, lParam);
}
[illegible]

    TRACE("Entering CPage2::DoDataExchange - %d\n",
```

```
        pDX->m_bSaveAndValidate);
    [illegible]
    [illegible]
    [illegible]
    [illegible]
}

[illegible]
[illegible]

/////////////////////////////////////////////////////////////////////////////
// [illegible]

/////////////////////////////////////////////////////////////////////////////
// [illegible]

[illegible]

[illegible]
{
    [illegible]
}

[illegible]
{
}

BOOL CPage3::OnCommand(WPARAM wParam, LPARAM lParam)
{
    SetModified(TRUE);
    return CPropertyPage::OnCommand(wParam, lParam);
}
[illegible]
{
    TRACE("Entering CPage3::DoDataExchange - %d\n",
        pDX->m_bSaveAndValidate);
    [illegible]
    [illegible]
}

[illegible]
[illegible]

/////////////////////////////////////////////////////////////////////////////
// [illegible]

/////////////////////////////////////////////////////////////////////////////
// [illegible]
```

см. след. стр.

```
[illegible]

[illegible]
{
    [illegible]
}

[illegible]
{
}

BOOL CPage4::OnCommand(WPARAM wParam, LPARAM lParam)
{
    SetModified(TRUE);
    return CPropertyPage::OnCommand(wParam, lParam);
}

[illegible]
{
    TRACE("Entering CPage4::DoDataExchange - %d\n",
        pDX->m_bSaveAndValidate);
    [illegible]
    [illegible]
    [illegible]
}

[illegible]
[illegible]

[illegible]
[illegible]

[illegible]
{
    [illegible]
    ((CSpinButtonCtrl*) GetDlgItem(IDC_SPIN1))->SetRange(8, 24);
    [illegible]
                    [illegible]
}

[illegible]
[illegible]

[illegible]
[illegible]
                    [illegible]
    [illegible]
{
}
```

```
CFontSheet::CFontSheet(LPCTSTR pszCaption, CWnd* pParentWnd,
                 UINT iSelectPage)
   :CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
   AddPage(&m_page1);
   AddPage(&m_page2);
   AddPage(&m_page3);
   AddPage(&m_page4);
}

CFontSheet::~CFontSheet()
{
}
BEGIN_MESSAGE_MAP(CFontSheet, CPropertySheet)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////
// CFontSheet message handlers
```

5. Добавьте в файл Ex12aView.h строку:

```
#include "Property.h"
```

6. **Добавьте две переменные и прототипы двух функций в класс *CEx12a-View*.** При использовании Class View директива *#include* для Property.h будет вставлена автоматически.

```
private:
   CFontSheet m_sh;
   BOOL m_bDefault; // TRUE - формат по умолчанию, FALSE - для выделения
```

Теперь добавьте прототип закрытой функции *Format*:

```
void Format(CHARFORMAT& cf);
```

Перед макросом *DECLARE_MESSAGE_MAP* разместите прототип защищенной функции *OnUserApply*:

```
afx_msg LRESULT OnUserApply(WPARAM wParam, LPARAM lParam);
```

7. **Отредактируйте код в файле Ex12aView.cpp.** Создайте обработчик пользовательского сообщения *WM_USERAPPLY*:

```
ON_MESSAGE(WM_USERAPPLY, OnUserApply)
```

Добавьте в функцию *OnCreate* прямо перед оператором *return 0* строки:

```
CHARFORMAT cf;
Format(cf);
m_rich.SetDefaultCharFormat(cf);
```

Модифицируйте конструктор объекта «вид», чтобы установить значения по умолчанию для переменных-членов окна свойств:

```
CEx12aView::CEx12aView() : m_sh("")
{
   m_sh.m_page1.m_nFont = 0;
   m_sh.m_page2.m_bBold = FALSE;
   m_sh.m_page2.m_bItalic = FALSE;
   m_sh.m_page2.m_bUnderline = FALSE;
   m_sh.m_page3.m_nColor = 0;
   m_sh.m_page4.m_nFontSize = 12;
   g_pView = this;
   m_bDefault = TRUE;
}
```

Отредактируйте обработчики команд форматирования:

```
void CEx12aView::OnFormatDefault()
{
   m_sh.SetTitle("Default Format");
   m_bDefault = TRUE;
   m_sh.DoModal();
}

void CEx12aView::OnFormatSelection()
{
   m_sh.SetTitle("Selection Format");
   m_bDefault = FALSE;
   m_sh.DoModal();
}

void CEx12aView::OnUpdateFormatSelection(CCmdUI* pCmdUI)
{
   long nStart, nEnd;
   m_rich.GetSel(nStart, nEnd);
   pCmdUI->Enable(nStart != nEnd);
}
```

Добавьте обработчик пользовательского сообщения *WM_USERAPPLY*:

```
LRESULT CEx12aView::OnUserApply(WPARAM wParam, LPARAM lParam)
{
   TRACE("CEx12aView::OnUserApply - wParam = %x\n", wParam);
   CHARFORMAT cf;
   Format(cf);
   if (m_bDefault) {
      m_rich.SetDefaultCharFormat(cf);
   }
   else {
      m_rich.SetSelectionCharFormat(cf);
   }
   return 0;
}
```

Вставьте, как показано ниже, вспомогательную функцию *Format*, чтобы инициализировать структуру *CHARFORMAT* в соответствии со значениями переменных-членов для окна свойств:

```
void CEx12aView::Format(CHARFORMAT& cf)
{
    cf.cbSize = sizeof(CHARFORMAT);
    cf.dwMask = CFM_BOLD | CFM_COLOR | CFM_FACE |
              CFM_ITALIC | CFM_SIZE | CFM_UNDERLINE;
    cf.dwEffects = (m_sh.m_page2.m_bBold ? CFE_BOLD : 0) |
                   (m_sh.m_page2.m_bItalic ? CFE_ITALIC : 0) |
                   (m_sh.m_page2.m_bUnderline ? CFE_UNDERLINE : 0);
    cf.yHeight = m_sh.m_page4.m_nFontSize * 20;
    switch(m_sh.m_page3.m_nColor) {
    case -1:
    case 0:
        cf.crTextColor = RGB(0, 0, 0);
        break;
    case 1:
        cf.crTextColor = RGB(255, 0, 0);
        break;
    case 2:
        cf.crTextColor = RGB(0, 255, 0);
        break;
    }
    switch(m_sh.m_page1.m_nFont) {
    case -1:
    case 0:
        strncpy(cf.szFaceName, "Times New Roman" ,LF_FACESIZE);
        break;
    case 1:
        strncpy(cf.szFaceName, "Arial" ,LF_FACESIZE);
        break;
    case 2:
        strncpy(cf.szFaceName, "Courier New" ,LF_FACESIZE);
        break;
    }
    cf.bCharSet = 0;
    cf.bPitchAndFamily = 0;
}
```

8. **Соберите и протестируйте усовершенствованное приложение Ex12a.** Введите какой-нибудь текст и вызовите из меню Format команду Default. Выбирая страницы в окне свойств и щелкая кнопку Apply, понаблюдайте за сообщениями, выводимыми операторами *TRACE* в окне Debug. Попробуйте выделить какой-нибудь текст и отформатировать выделенный фрагмент.

Обработка кнопки Apply

Вас может удивить, какой способ обработки кнопки Apply используют классы окна свойств. Переопределенная виртуальная функция *OnCommand* во всех классах

страниц свойств активизирует кнопку Apply всякий раз, когда какой-либо элемент управления посылает сообщение данной странице. Это отлично срабатывает для страниц 1–3 в примере Ex12a, но не для страницы 4, где *OnCommand* вызывается в процессе начального обмена данными между элементом управления «наборный счетчик» и связанным с ним полем ввода.

Переопределенная виртуальная функция *OnApply* в классе *CPage1* посылает объекту «вид» пользовательское сообщение. Поиск этого объекта осуществляется довольно грубым способом — через глобальную переменную, установленную классом «вид». Лучше передавать указатель на объект «вид» конструктору окна свойств, а затем конструктору страницы свойств.

Класс «вид» вызывает функцию *DoModal* окна свойств как для форматирования по умолчанию, так и для форматирования выделенного фрагмента. Для указания режима служит флаг *m_bDefault*. Проверять возвращаемое значение *DoModal* не нужно, так как пользовательское сообщение отправляется при щелчке как кнопки OK, так и Apply. При щелчке пользователем кнопки Cancel сообщение вообще не посылается.

Класс *CMenu*

До этого момента каркас приложений и редактор меню скрывали от нас класс меню *CMenu*. Объект этого класса способен представлять любое меню Windows, в том числе меню верхнего уровня и связанные с ними подменю. Чаще всего, когда вызывается функция *Create* или *LoadFrame* для окна-рамки и объект *CMenu* явным образом не создается, ресурс меню подсоединяется прямо к этому окну. Функция-член *GetMenu* класса *CWnd* возвращает указатель на временный объект *CMenu*. Получив этот указатель, вы обретаете свободный доступ к меню и можете изменять его.

Допустим, вы хотите переключать меню после запуска приложения. Идентификатор у исходного меню — всегда *IDR_MAINFRAME*. Если нужно второе меню, его создают в редакторе меню и определяют нужный идентификатор. Затем в программе создается объект *CMenu*, загружается меню из ресурса с помощью *CMenu::LoadMenu* и вызывается *CWnd::SetMenu*, чтобы присоединить новое меню к окну-рамке. Затем вызывается метод *Detach*, чтобы отсоединить от объекта описатель *HMENU*; тогда созданное меню не уничтожается при выходе объекта *CMenu* из области видимости.

Можно определить меню в ресурсе и потом модифицировать его в период выполнения программы. В период выполнения можно создать и все меню целиком, не используя ресурсов. В любом случае для этого пригодятся такие функции-члены класса *CMenu*, как *ModifyMenu*, *InsertMenu* и *DeleteMenu*. Каждая из них работает с отдельным элементом меню, определяемым по идентификатору или по индексу его позиции в меню.

Объект «меню» в действительности состоит из вложенной структуры подменю. Функция-член *GetSubMenu* возвращает указатель на объект *CMenu*, который представляет раскрывающееся меню в главном объекте *CMenu*. Функция *CMenu::GetMenuString* возвращает текст в элементе меню, указанному либо по индексу, либо по идентификатору команды. В последнем случае выполняется поиск по всей системе меню, включая вложенные.

Создание контекстных меню

Контекстные меню (floating shortcut menu) — одна из последних новаций в пользовательском интерфейсе. Щелчок правой кнопкой открывает меню с командами, которые относятся к текущему контексту. Такие меню легко создать с помощью графического редактора и MFC-функции *CMenu::TrackPopupMenu*.

1. Откройте редактор меню и добавьте в файл ресурсов своего проекта новое пустое меню.
2. Введите имя для крайнего слева элемента верхнего уровня и добавьте команды в получившееся контекстное меню.
3. В окне Properties средства Class View, создайте обработчик сообщения *WM_CONTEXTMENU* в классе «вид» или в классе другого окна, получающего сообщения от кнопок мыши, и запрограммируйте его так:

```
void CMyView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    CMenu menu;
    menu.LoadMenu(IDR_MYFLOATINGMENU);
    menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN |
        TMP_RIGHTBUTTON, point.x, point.y, this);
}
```

Команды контекстного меню можно сопоставить обработчикам при помощи окна Properties в Class View тем же способом, что и команды в меню окна-рамки.

Расширенная обработка команд

Кроме макроса таблицы сообщений *ON_COMMAND*, в библиотеке MFC есть его расширенный вариант *ON_COMMAND_EX*. Он обеспечивает две возможности, которые не поддерживаются для обычного командного сообщения: параметр обработчика, задающий идентификатор команды, и возможность отказаться от обработки команды в период выполнения, отослав ее следующему объекту в пути маршрутизации команд. Если расширенный обработчик команды возвращает *TRUE*, команда считается обработанной, а если *FALSE* — каркас приложений ищет следующий обработчик.

Параметр — идентификатор команды полезен, если надо обрабатывать одной функцией несколько связанных друг с другом командных сообщений. Наверняка вы найдете способы применить эту возможность.

Мастер окна Class View не поможет в работе с расширенными обработчиками команд, так что придется кодировать их самостоятельно (за скобками *AFX_MSG_MAP*). Допустим, *IDM_ZOOM_1* и *IDM_ZOOM_2* — идентификаторы взаимосвязанных команд, определенные в файле Resource.h. Тогда для обработки обоих сообщений одной функцией *OnZoom* надо сделать так:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_COMMAND_EX(IDM_ZOOM_1, OnZoom)
    ON_COMMAND_EX(IDM_ZOOM_2, OnZoom)
END_MESSAGE_MAP()
```

```
BOOL CMyView::OnZoom(UINT nID)
{
    if (nID == IDM_ZOOM_1) {
        // код для первой команды
    }
    else {
        // код для второй команды
    }
        // код, общий для обеих команд
    return TRUE; // обработка команды завершена
}
```

А вот прототип функции:

```
afx_msg BOOL OnZoom(UINT nID);
```

Есть и другие макросы таблицы сообщений MFC, полезные при обработке групп команд. Такая обработка может понадобиться в приложениях, в которых есть динамические меню. Вот эти макросы: ON_COMMAND_RANGE, ON_COMMAND_EX_RANGE и ON_UPDATE_COMMAND_UI_RANGE

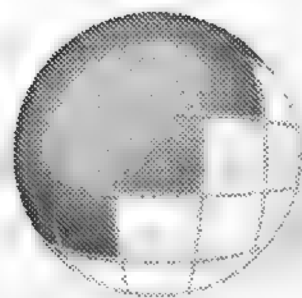
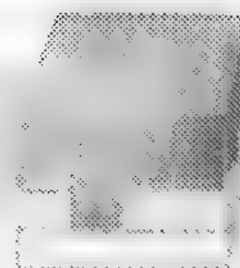
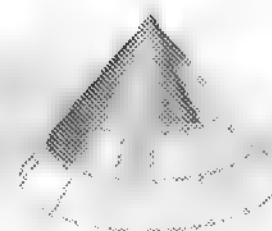
Если бы значения *IDM_ZOOM_1* и *IDM_ZOOM_2* были последовательными, то таблицу сообщений *CMyView* можно было бы переписать:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_COMMAND_EX_RANGE(IDM_ZOOM_1, IDM_ZOOM_2, OnZoom)
END_MESSAGE_MAP()
```

Теперь *OnZoom* будет вызываться при выборе обоих элементов меню; обработчик может определить выбранную команду по значению целочисленного параметра.

ГЛАВА

13

Панели инструментов и строки состояния

Во всех приведенных до этого примерах программ на Visual C++ имелись панели инструментов и строки состояния. MFC Application Wizard генерирует код, инициализирующий эти элементы каркаса приложений, если только явно не отказаться от предлагаемых по умолчанию параметров Dockable Toolbar и Initial Status Bar. Созданная по умолчанию панель инструментов содержит графические эквиваленты многих стандартных элементов меню, формируемых каркасом приложений; в строке состояния по умолчанию отображаются подсказки для команд меню и индикаторы состояния переключателей клавиатуры: CAPS, NUM и SCRL.

В этой главе вы узнаете, как настроить для своей программы панель инструментов и строку состояния, научитесь добавлять на панель инструментов собственные графические кнопки и управлять их отображением. Кроме того, вы научитесь отключать отображение подсказок и индикаторов клавиатуры в строке состояния, что позволит вашей программе самостоятельно управлять этой строкой.

Панели элементов управления и каркас приложений

Панель инструментов (toolbar) — это объект класса *CToolBar*, а *строка состояния* (status bar) — объект класса *CStatusBar*. Оба класса производные от *CControlBar*, а тот в свою очередь — от *CWnd*. Класс *CControlBar* поддерживает окна панелей управления, размещаемых в окне-рамке. Окна этих панелей автоматически изменяют свои размеры и положение при масштабировании и перемещении окна-рамки. Каркас приложений создает и удаляет эти объекты и их окна. MFC Application Wizard генерирует код панелей элементов управления для производного класса окна-рамки, хранящегося в файлах MainFrm.cpp и MainFrm.h.

В типичном SDI-приложении объект *CToolBar* занимает верхнюю, а объект *CStatusBar* — нижнюю часть клиентской области *CMainFrame*. Между ними располагается окно представления.

С версии 4.0 в MFC-библиотеке панель инструментов формируется на основе стандартного элемента управления *панель инструментов* (toolbar common control), появившегося в Windows 95, и поэтому поддерживает *стыковку* (docking) с окном-рамкой. Хотя программный интерфейс остался практически таким же, что и в предыдущих версиях MFC, работать с изображениями кнопок стало легче, так как редактор ресурсов поддерживает соответствующий специальный тип ресурсов.

Если MFC Application Wizard сгенерировал код панелей элементов управления для приложения, пользователь получает возможность включать и отключать отображение этих панелей из меню View. При отключении панель исчезает, а размер окна представления корректируется. Несмотря на все общие (только что описанные) особенности, панель инструментов и строка состояния независимы друг от друга и имеют разные характеристики.

В Visual C++ 6.0 введена новая панель инструментов — *rebar*, основанная на элементах управления из состава Microsoft Internet Explorer. Она предоставляет *скользящий* (sliding) стиль оформления, характерный для Internet Explorer. Мы познакомимся с ней чуть позже.

Панели инструментов

Панель инструментов — это строка горизонтально (или вертикально) расположенных графических кнопок, иногда объединенных в группы. Разбивка на группы определяется логикой приложения. Изображения кнопок хранятся в общем растровом изображении из состава ресурсов приложения. При щелчке кнопки она отправляет командное сообщение — так же, как меню и быстрые клавиши. Для обновления состояния кнопок служат обработчики обновления командного пользовательского интерфейса, вызываемые каркасом приложений для изменения внешнего вида кнопок.

Растровое изображение панели инструментов

Может показаться, что у каждой кнопки на панели инструментов свое растровое изображение, но на самом деле оно одно на всю панель. Каждой кнопке на нем выделяется ячейка высотой 15 и шириной 16 пикселов. Каркас приложений обеспечивает прорисовку границ кнопок и изменяет их вместе с цветом фона кнопки, отображая ее текущее состояние. На рис. 13-1 показано растровое изображение и соответствующая ему панель инструментов.

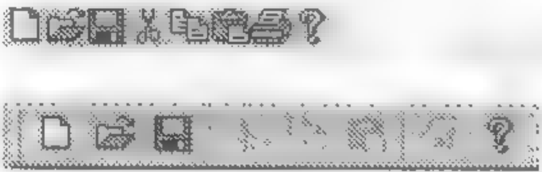


Рис. 13-1. *Растровое изображение и соответствующая панель инструментов*

Растровое изображение панели инструментов хранится в файле Toolbar.bmp в подкаталоге RES приложения. В RC-файле оно идентифицируется как *IDR_MAIN-*

FRAME. Это растровое изображение напрямую не изменяют — вместо этого используют специальные средства графического редактора Microsoft Visual Studio.

Состояния кнопок панели инструментов

Кнопка может находиться в одном из нескольких состояний (табл. 13-1). (В более поздних версиях панелей инструментов есть несколько дополнительных состояний.)

Табл. 13-1. Состояния кнопок

Состояние	Описание
0	Нормальное — «отжата»
TBSTATE_CHECKED	Помечена («нажата»)
TBSTATE_ENABLED	Доступна; если это состояние не установлено, кнопка недоступна и изображение на ней блеклое
TBSTATE_HIDDEN	Невидима
TBSTATE_INDETERMINATE	Недоступна (блеклая)
TBSTATE_PRESSED	Выбрана («нажата») мышью
TBSTATE_WRAP	Кнопка является последней в строке

Кнопка может быть командной (pushbutton) (нажата только при щелчке, а в нормальном состоянии отжата) или флажком (check box button) (сохраняет состояние, нажимается и отжимается щелчком). На стандартной панели инструментов, формируемой каркасом приложений, все кнопки — командные.

Панель инструментов и командные сообщения

Когда пользователь щелкает кнопку на панели инструментов, генерируется командное сообщение. Оно маршрутизируется так же, как командные сообщения от меню (см. главу 12). Чаще всего кнопка на панели инструментов дублирует команду меню. Так, кнопка с изображением дискеты на стандартной панели инструментов, формируемой каркасом приложений, эквивалентна команде Save меню File. Обе генерируют команду *ID_FILE_SAVE*. А объекту, получившему командное сообщение, безразлично, как оно сгенерировано: щелчком кнопки на панели инструментов или выбором команды из меню.

Однако кнопки панели инструментов не обязаны дублировать меню — рекомендуется определить для кнопки быструю клавишу, чтобы пользователь мог выполнить команду с клавиатуры или через Windows-макрос. Для определения обработчиков командных сообщений и сообщений обновления командного интерфейса служат окно Properties и Class View независимо от того, соответствуют ли кнопки панели инструментов командам меню.

С панелью инструментов связан ресурс растрового изображения и сопутствующий ресурс панели инструментов в RC-файле, который определяет команды меню, связанные с кнопками. У растрового изображения и ресурса панели инструментов одинаковый идентификатор — обычно *IDR_MAINFRAME*. Так выглядит содержимое ресурса панели инструментов, сгенерированное мастером MFC Application Wizard:

```
IDR_MAINFRAME TOOLBAR  16, 15
BEGIN
    BUTTON ID_FILE_NEW
    BUTTON ID_FILE_OPEN
    BUTTON ID_FILE_SAVE
        SEPARATOR
    BUTTON ID_EDIT_CUT
    BUTTON ID_EDIT_COPY
    BUTTON ID_EDIT_PASTE
        SEPARATOR
    BUTTON ID_FILE_PRINT
    BUTTON ID_APP_ABOUT
END
```

Константы *SEPARATOR* служат для отделения групп кнопок друг от друга. Если число кнопок в растровом изображении панели инструментов превосходит количество элементов в ресурсе (без учета разделителей), лишние кнопки просто не отображаются.

При редактировании панели инструментов в редакторе ресурсов модифицируется и ресурс растрового изображения, и ресурс панели инструментов. Чтобы отредактировать свойства кнопки (в том числе идентификатор), выберите ее изображение, а затем в окне Properties определите ее свойства.

Обновление пользовательского интерфейса для панелей инструментов

Как вы помните из главы 12, обработчики сообщений обновления пользовательского интерфейса служат для отключения команд меню или отметки их галочками. Эти же обработчики применяются и для кнопок на панели инструментов. Если подобный обработчик вызывает функцию-член *CCmdUI::Enable* с параметром *FALSE*, соответствующая кнопка отключается (становится блеклой) и перестает реагировать на щелчки.

Функция *CCmdUI::SetCheck* ставит рядом с командой меню галочку, а на панели инструментов реализует кнопку-флажок. После вызова *SetCheck* с параметром 1 кнопка «залипает» в нажатом состоянии, а с параметром 0 возвращается в исходное, отжатое состояние.

Примечание Если параметр функции *SetCheck* равен 2, кнопка устанавливается в *неопределенное* (indeterminate) состояние — она выглядит как отключенная, но по-прежнему активна, и ее цвет несколько ярче.

Обработчики сообщений обновления пользовательского интерфейса для команд меню вызываются только при прорисовке элементов соответствующего меню. Но панель инструментов постоянно на экране — спрашивается, когда же вызывать обработчики обновления интерфейса? Они вызываются в моменты простоя приложения, чтобы состояние кнопок можно было постоянно обновлять. Если для элемента меню и кнопки на панели инструментов используется один обработчик, он вызывается и в момент простоя, и при открытии меню.

Всплывающие подсказки

Всплывающие подсказки (tooltips) встречаются в разных Windows-приложениях, в том числе и в самой Visual Studio. Если указатель мыши задержать на кнопке панели инструментов, то вскоре рядом с кнопкой появится маленькое окошко с текстом. В главе 12 мы уже говорили, что элементам меню можно сопоставлять строки подсказки — элементы строкового ресурса с соответствующими идентификаторами. Чтобы создать всплывающую подсказку, просто добавьте ее в конец подсказки для меню, начав текст с символа новой строки (\n). Редактор ресурсов позволяет модифицировать строку подсказки при редактировании изображений кнопок на панели инструментов. Для этого просто выберите изображение панели инструментов и в окне Properties отредактируйте свойство Prompt.

Поиск основного окна-рамки

Объекты панели инструментов и строки состояния, с которыми вам предстоит работать, связаны с основным окном-рамкой приложения, а не окном представления. Как же найти основное окно-рамку? В SDI-приложении для этого служит функция *CWnd::GetParentFrame*. К сожалению, она не работает в MDI-приложении, так как в этом случае родительской рамкой будет дочернее, а не основное окно-рамка.

Чтобы класс «вид» работал как в SDI-, так и в MDI-приложениях, нужно искать основное окно-рамку через объект-приложение. Указатель на этот объект возвращает глобальная функция *AfxGetApp*; такой указатель позволяет обратиться и к элементу данных *m_pMainWnd* класса *CWinApp*[1]. В MDI-приложении код, устанавливающий *m_pMainWnd*, генерирует MFC Application Wizard, но в SDI-приложении эта переменная-член определяется каркасом приложений при формировании объекта «вид». Инициализировав *m_pMainWnd*, можно использовать его в классе «вид» для доступа к панели инструментов в окне-рамке, например, так:

```
CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
CToolBar* pToolBar = &pFrame->m_wndToolBar;
```

Примечание Переменную *m_pMainWnd* следует привести из типа *CFrameWnd** в *CMainFrame**, так как *m_wndToolBar* — переменная-член именно этого производного класса. Вам также придется сделать *m_wndToolBar* открытой переменной или объявить свой класс дружественным классу *CMainFrame*.

Аналогичную технику можно применять для поиска объектов меню, строк состояния и диалоговых окон.

Примечание В SDI-приложении значение *m_pMainWnd* еще не установлено при вызове обработчика сообщений *OnCreate* класса «вид». Для доступа к основному окну-рамке в *OnCreate* нужна функция *GetParentFrame*.

---

[1] Проще воспользоваться MFC-функцией *AfxGetMainWnd*. — *Прим. перев.*

Пример Ex13a: работа с панелями инструментов

Мы создадим три специальных кнопки для управления рисованием в окне представления и меню Draw с тремя элементами, соответствующими этим кнопкам.

Команда меню	Назначение
Circle	Рисует круг в окне представления.
Square	Рисует квадрат в окне представления.
Pattern	Включает/отключает наклонную штриховку новых кругов и квадратов.

Меню и панель инструментов позволяют пользователю попеременно рисовать круги и квадраты. После рисования круга отключается команда Circle и соответствующая кнопка на панели инструментов, а после рисования квадрата — команда и кнопка Square. Если включена штриховка, команда Pattern в меню Draw отмечена галочкой, а одноименная кнопка на панели инструментов переводится в нажатое состояние.

На рис. 13-2 показана программа в действии. Пользователь только что нарисовал заштрихованный квадрат. Обратите внимание на состояние трех кнопок, связанных с рисованием.

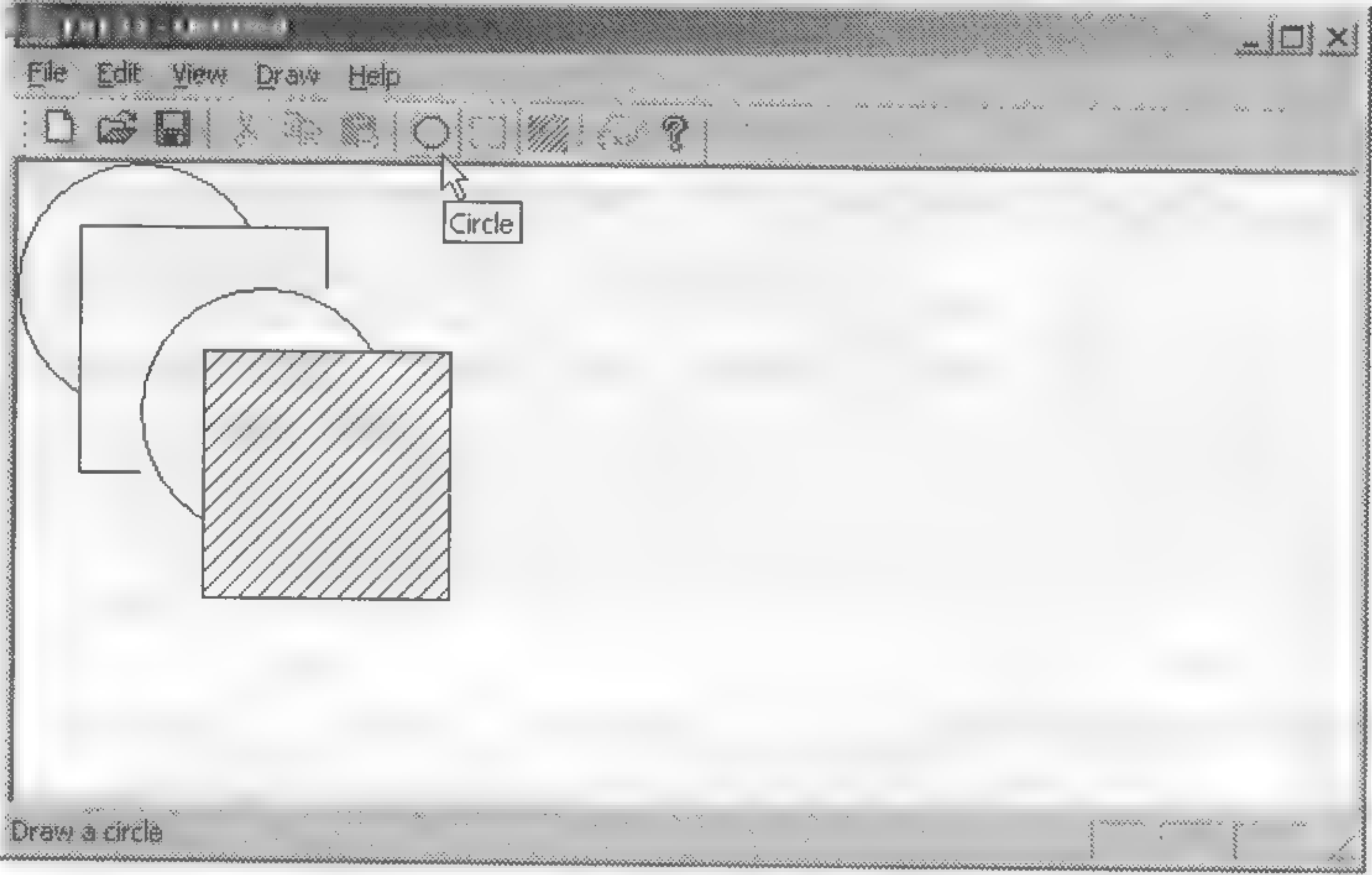
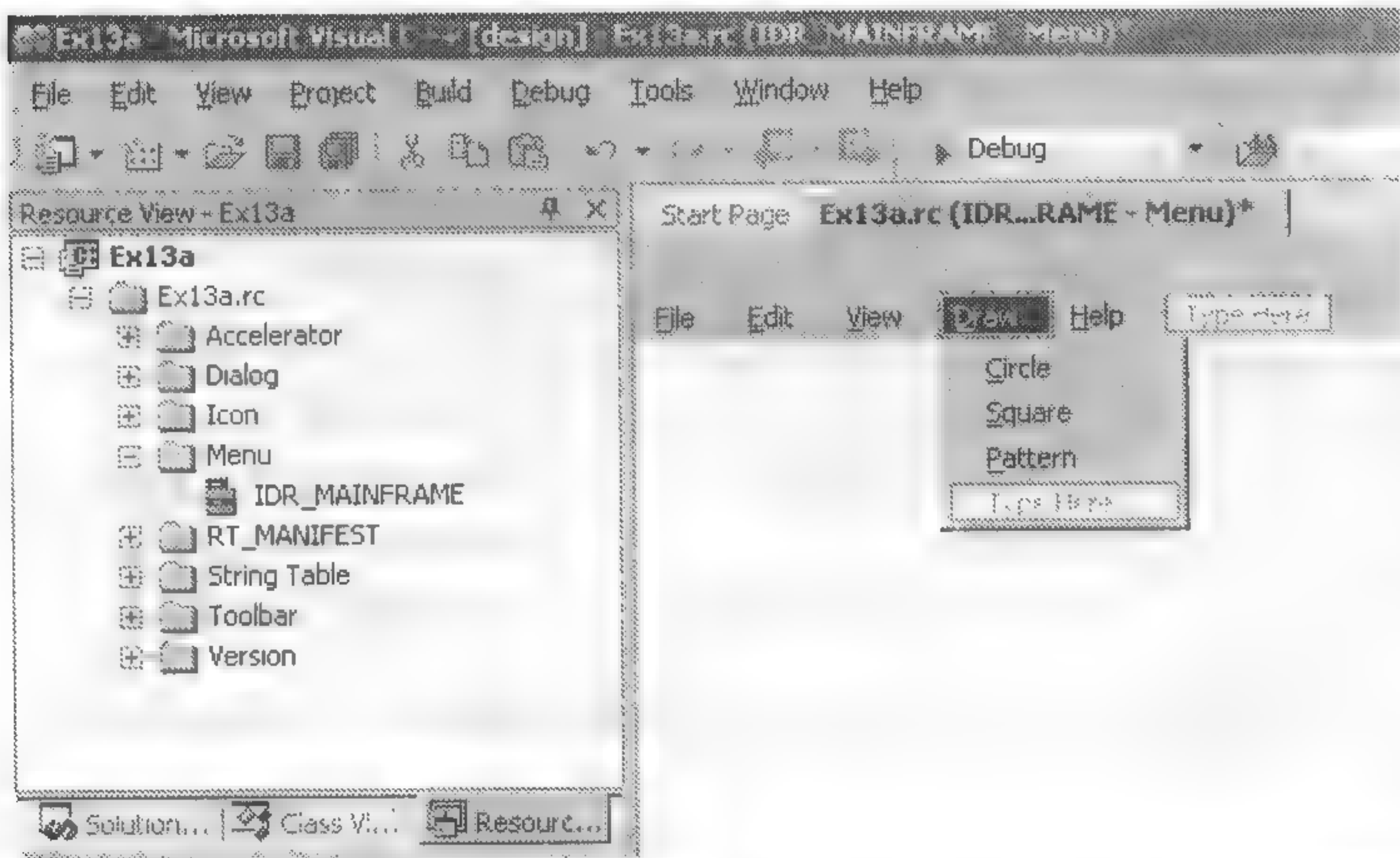


Рис. 13-2. *Программа Ex13a в действии*

Пример Ex13a знакомит вас с возможностями редактора ресурсов при работе с панелями инструментов. Кодировать на C++ придется совсем немного.

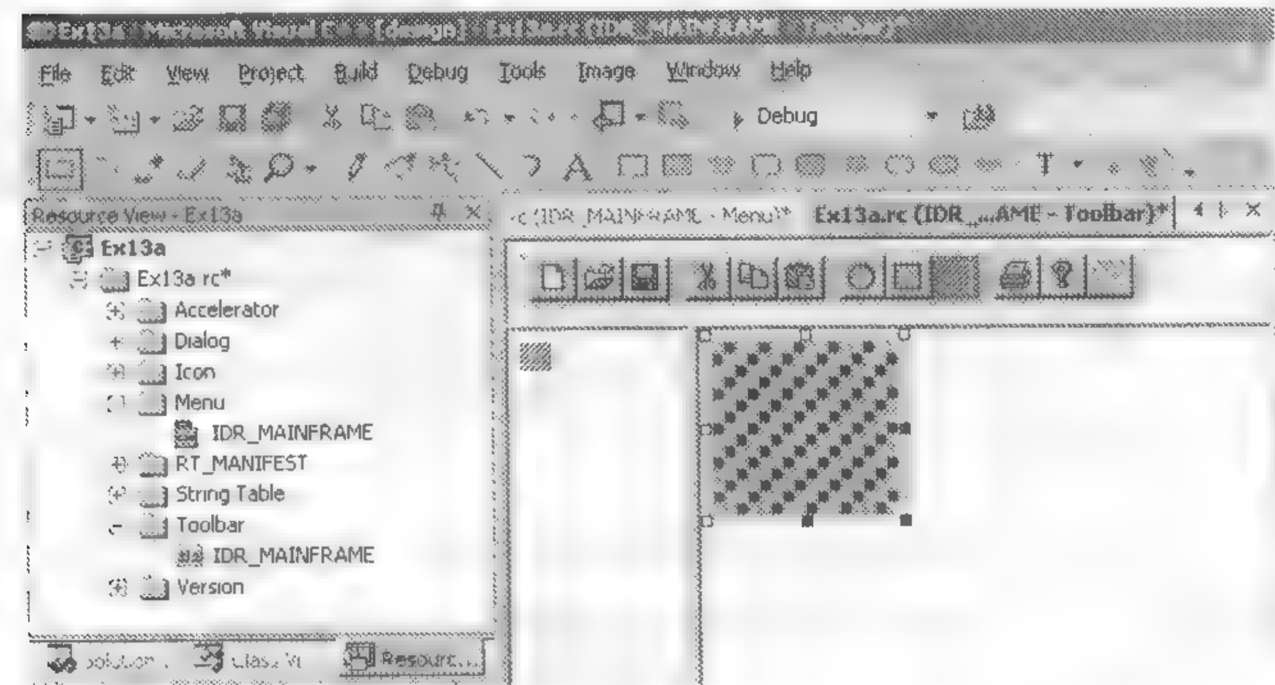
1. **Средствами MFC Application Wizard создайте проект Ex13a.** Последовательно выберите в меню File команды New и Project. В качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex13a. На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **В редакторе ресурсов измените основное меню приложения.** В окне Resource View дважды щелкните значок *IDR_MAINFRAME* в разделе Menu. Отредактируйте ресурс меню *IDR_MAINFRAME*, создав меню Draw (чтобы изменить положение элемента меню, его достаточно перетащить):



В окне Properties назначьте новым элементам меню идентификаторы команд:

Заголовок (Caption)	Идентификатор команды	Строка подсказки (Promt)
&Circle	*ID_DRAW_CIRCLE*	Draw a circle\nCircle
&Square	*ID_DRAW_SQUARE*	Draw a square\nSquare
&Pattern	*ID_DRAW_PATTERN*	Change the pattern\nPattern

3. **Используя редактор ресурсов, измените панель инструментов.** Отредактируйте ресурс растрового изображения *IDR_MAINFRAME*, создав новую группу из трех кнопок:



Редактор панелей инструментов интуитивно понятен: новые кнопки добавляются перетаскиванием на правый край панели. Нарисуйте изображение на кнопке инструментами Ellipsis, Rectangle и Line панели инструментов Image Editor. Чтобы добавить разделитель, перетащите и бросьте кнопку чуть левее того места, где он должен быть, — группа сместится и отделится от остальных кнопок. Клавиша Del стирает пикселы на изображении кнопок. Чтобы удалить кнопку, переместите ее за пределы панели инструментов.

В окне Properties присвойте новым кнопкам идентификаторы *ID_DRAW_CIRCLE*, *ID_DRAW_SQUARE* и *ID_DRAW_PATTERN*.

4. **Создайте в классе CEx13aView обработчики сообщений.** Выберите класс CEx13aView в окне Class View, в окне Properties щелкните кнопку Events и до-

бавьте обработчики командных сообщений и сообщений обновления пользовательского интерфейса:

Идентификатор объекта	**Сообщение**	**Функция-член**
ID_DRAW_CIRCLE	*COMMAND*	*OnDrawCircle*
ID_DRAW_CIRCLE	*UPDATE_COMMAND_UI*	*OnUpdateDrawCircle*
ID_DRAW_PATTERN	*COMMAND*	*OnDrawPattern*
ID_DRAW_PATTERN	*UPDATE_COMMAND_UI*	*OnUpdateDrawPattern*
ID_DRAW_SQUARE	*COMMAND*	*OnDrawSquare*
ID_DRAW_SQUARE	*UPDATE_COMMAND_UI*	*OnUpdateDrawSquare*

5. **Добавьте в класс CEx13aView три переменные-члены.** Вручную отредактируйте файл Ex13aView.h:

```
protected:
    CRect m_rect;
    BOOL  m_bCircle;
    BOOL  m_bPattern;
```

6. **Отредактируйте файл Ex13aView.cpp.** Конструктор *CEx13aView* просто инициализирует переменные-члены класса. Добавьте выделенный код:

```
CEx13aView::CEx13aView() : m_rect(0, 0, 100, 100)
{
    m_bCircle = TRUE;
    m_bPattern = FALSE;
}
```

В зависимости от значения флажка *m_bCircle* функция *OnDraw* рисует эллипс или квадрат. По значению *m_bPattern* выбирается кисть: белая или с наклонной штриховкой:

```
void CEx13aView::OnDraw(CDC* pDC)
{
    Cex13aDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CBrush brush(HS_BDIAGONAL, 0L); // кисть с диагональной штриховкой

    if (m_bPattern) {
        pDC->SelectObject(&brush);
    }
    else {
        pDC->SelectStockObject(WHITE_BRUSH);
    }
    if (m_bCircle) {
        pDC->Ellipse(m_rect);
    }
    else {
        pDC->Rectangle(m_rect);
    }
```

```
    pDC->SelectStockObject(WHITE_BRUSH);    // Если переменная установлена,
                                            // кисть сбрасывается
}
```

OnDrawCircle обрабатывает командное сообщение *ID_DRAW_CIRCLE*, а *OnDrawSquare* — сообщение *ID_DRAW_SQUARE*. Обе функции перемещают прямоугольную область рисования вниз и вправо, потом объявляют ее недействительной, и она перерисовывается функцией *OnDraw*. В результате окружности и квадраты поочередно выводятся в окно по диагонали. Вывод на дисплей не буферизуется, так что нарисованные фигуры не перерисовываются, когда окно перекрывается другим или минимизируется.

```
void CEx13aView::OnDrawCircle()
{
    m_bCircle = TRUE;
    m_rect += CPoint(25, 25);
    InvalidateRect(m_rect);
}

void CEx13aView::OnDrawSquare()
{
    m_bCircle = FALSE;
    m_rect += CPoint(25, 25);
    InvalidateRect(m_rect);
}
```

Следующие два обработчика, обновляющие пользовательский интерфейс, поочередно включают и отключают в меню команды Circle и Square и соответствующие кнопки. В любой момент доступна только одна из двух возможностей.

```
void CEx13aView::OnUpdateDrawCircle(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!m_bCircle);
}

void CEx13aView::OnUpdateDrawSquare(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_bCircle);
}
```

Функция *OnDrawPattern* меняет состояние флажка *m_bPattern* на противоположное:

```
void CEx13aView::OnDrawPattern()
{
    m_bPattern ^= 1;
}
```

Обработчик *OnUpdateDrawPattern* обновляет элемент меню и кнопку Pattern в соответствии с состоянием флажка *m_bPattern*: кнопка «залипает» или возвращается в исходное состояние, а рядом с командой меню появляется или исчезает галочка.

```
void CEx13aView::OnUpdateDrawPattern(CCmdUI* pCmdUI)
{
   pCmdUI->SetCheck(m_bPattern);
}
```

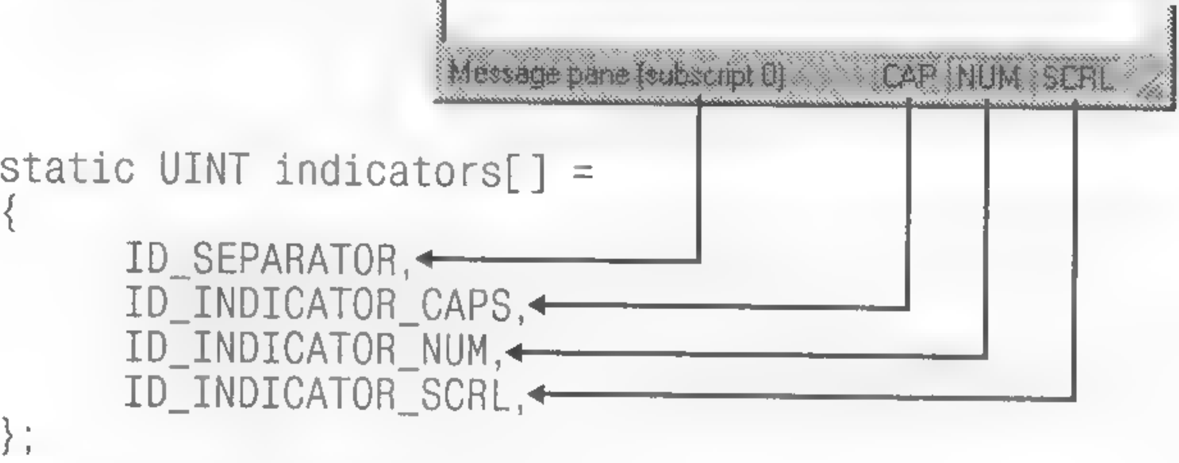
7. **Соберите и протестируйте приложение Ex13a.** Обратите внимание на поведение кнопок на панели инструментов. Поэкспериментируйте с элементами меню и заметьте, что они меняют свое состояние в соответствии с состоянием приложения. Понаблюдайте также за появлением всплывающей подсказки, когда курсор мыши «замирает» над одной из кнопок на панели инструментов.

Строка состояния

Окно строки состояния не принимает информации от пользователя и не генерирует командных сообщений. Его задача — просто показывать под управлением программы текст в соответствующих секциях. Строка состояния поддерживает два типа текстовых секций: строку сообщений и индикаторы. Чтобы вывести информацию в строке состояния, сначала отключите стандартную строку состояния, которая отображает подсказки по элементам меню и сообщает статус клавиатуры.

Определение секций в строке состояния

Секции в строке состояния определяются статическим массивом *indicators*, который MFC Application Wizard создает в файле MainFrm.cpp. Константа *ID_SEPARATOR* указывает на секцию для строки сообщений, а другие константы служат идентификаторами строковых ресурсов и определяют секции индикаторов. Взаимосвязь массива *indicators* и стандартной строки состояния выглядит так (рис. 13-3):




```
static UINT indicators[] =
{
     ID_SEPARATOR,
     ID_INDICATOR_CAPS,
     ID_INDICATOR_NUM,
     ID_INDICATOR_SCRL,
};
```

Рис. 13-3. *Строка состояния и массив* indicators

Функция-член *CStatusBar::SetIndicators*, вызываемая в производном классе окна-рамки приложения, приводит строку состояния в соответствие с содержимым массива *indicators*.

Строка сообщений

В этой секции отображается строка, динамически определяемая программой. Чтобы определить выводимый текст, надо получить доступ к объекту строки состояния, после чего вызвать функцию-член *CStatusBar::SetPaneText*, передав ей индекс секции. Индексы начинаются с 0; нулевая секция — крайняя слева, секция 1 размещается правее и т. д.

Приведенный ниже фрагмент кода входит в функцию-член класса «вид». Здесь приходится сначала подниматься на уровень объекта-приложения, а потом возвращаться к основному окну-рамке:

```
CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
CStatusBar* pStatus = &pFrame->m_wndStatusBar;
pStatus->SetPaneText(0, "Строка сообщения в первой секции");
```

Обычно длина секции сообщений составляет ровно четверть ширины экрана. Однако у первой секции (индекс 0) длина переменная: она не менее четверти ширины экрана и может увеличиваться, если в строке состояния есть место.

Индикатор состояния

Секция индикатора состояния связана с единственной строкой ресурса, которая отображается или скрывается в соответствии с логикой функции-обработчика. Индикатор обозначается идентификатором строкового ресурса, и тот же идентификатор служит для маршрутизации сообщений обновления интерфейса. Индикатор Caps Lock обрабатывается в классе окна-рамки при помощи приведенных ниже записи таблицы сообщений и функции-обработчика (функция *Enable* включает индикатор, если активен режим Caps Lock):

```
ON_UPDATE_COMMAND_UI(ID_INDICATOR_CAPS, OnUpdateKeyCapsLock)

void CMainFrame::OnUpdateKeyCapsLock(CCmdUI* pCmdUI)
{
   pCmdUI->Enable(::GetKeyState(VK_CAPITAL) & 1);
}
```

Функции, обновляющие пользовательский интерфейс, вызываются в цикле простоя приложения, поэтому строка состояния обновляется всякий раз, когда приложение получает сообщения. Длина секции индикатора равна длине соответствующей строки из ресурса.

Управление строкой состояния

Стандартной строке состояния присваивается идентификатор дочернего окна *AFX_IDW_STATUS_BAR*. Именно его каркас приложений ищет для отображения подсказки по элементам меню. Обработчики сообщений обновления пользовательского интерфейса применяют три идентификатора строковых ресурсов для индикаторов состояния клавиатуры в базовом классе окна-рамки: *ID_INDICATOR_CAPS*, *ID_INDICATOR_NUM* и *ID_INDICATOR_SCRL*. Чтобы перехватить управление строкой состояния, нужно назначить другой идентификатор дочернего окна и другие константы для индикаторов.

Примечание Изменять идентификатор дочернего окна строки состояния имеет смысл, только если нужно предотвратить вывод каркасом приложений подсказок в секцию 0. Если подсказки вас устраивают, можете не читать приведенные далее инструкции.

Идентификатор для окна строки состояния назначается вызовом *CStatusBar::Create* в функции-члене *OnCreate* производного класса окна-рамки. Эта функция содержится в файле MainFrm.cpp, генерируемом MFC Application Wizard. Идентификатор окна — третий параметр функции *Create* и по умолчанию равен *AFX_IDW_STATUS_BAR*.

Чтобы назначить свой идентификатор, замените вызов:

```
m_wndStatusBar.Create(this);
```

на:

```
m_wndStatusBar.Create(this, WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, ID_MY_STATUS_BAR);
```

Конечно, нужно определить и константу *ID_MY_STATUS_BAR* в файле resource.h, применяя редактор символов.

Но мы кое-что забыли. Стандартное меню View, формируемое каркасом приложений, позволяет включать и отключать отображение строки состояния. Эта логика реализуется кодом, связанным с идентификатором окна *AFX_IDW_STATUS_BAR*, который тоже придется изменить. В своем производном классе окна-рамки напишите элементы таблицы сообщений и обработчики для команды *ID_VIEW_STATUS_BAR* и сообщений, связанных с обновлением пользовательского интерфейса. *ID_VIEW_STATUS_BAR* — это идентификатор элемента меню Status Bar. Обработчики в производном классе переопределяют стандартные обработчики базового класса *CFrameWnd*. Подробности — в примере Ex13b.

Пример Ex13b: строка состояния

Здесь стандартная строка состояния заменяется новой с текстовыми секциями:

Индекс секции	Идентификатор строки	Тип	Описание
0	*ID_SEPARATOR* (0)	Строка сообщений	*x*-координата курсора
1	*ID_SEPARATOR* (0)	Строка сообщений	*y*-координата курсора
2	*ID_INDICATOR_LEFT*	Индикатор состояния	Состояние левой кнопки мыши
3	*ID_INDICATOR_RIGHT*	Индикатор состояния	Состояние правой кнопки мыши

Взгляните на новую строку состояния: при увеличении размеров окна-рамки крайняя левая секция растягивается за пределы своей обычной длины (рис. 13-4):

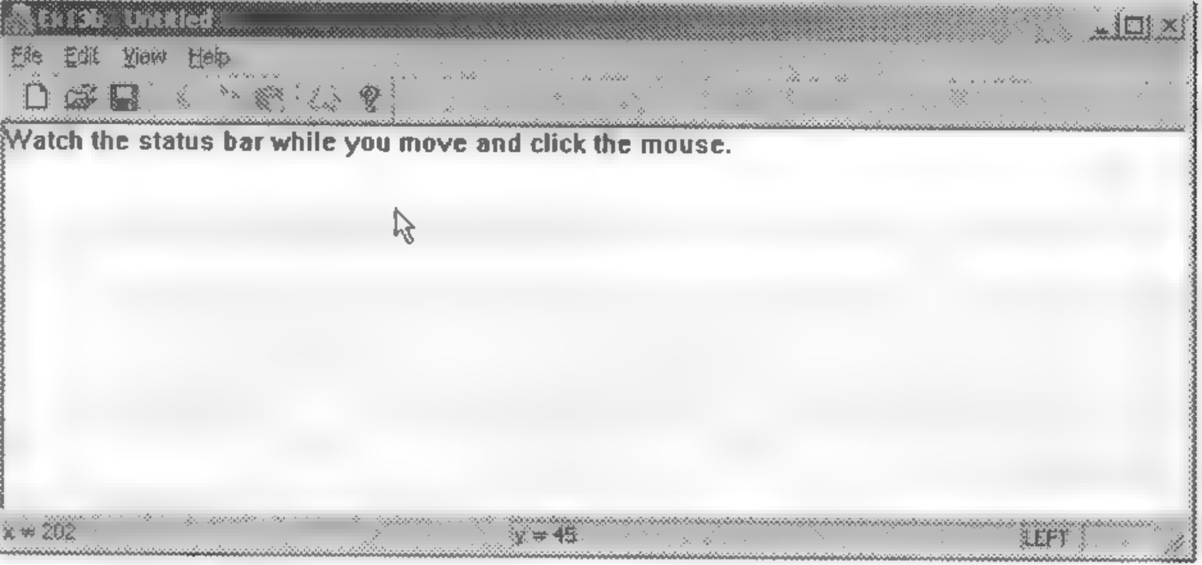


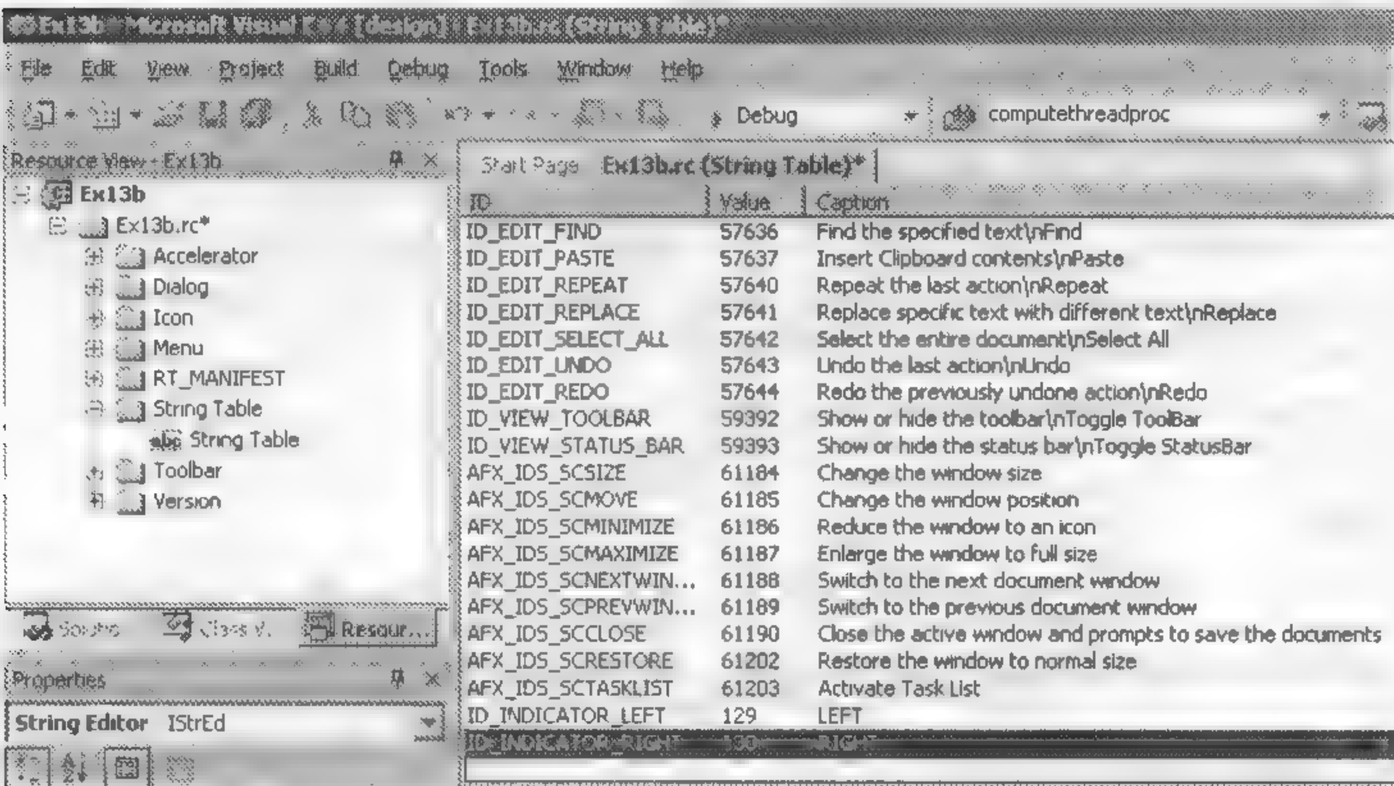
Рис. 13-4. *Строка состояния в программе Ex13b*

Итак, создаем программу Ex13b.

1. **Средствами MFC Application Wizard создайте проект Ex13b.** Последовательно выберите в меню File команды New и Project. В качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex13b. На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальных параметров не меняйте.
2. **Отредактируйте в редакторе ресурсов ресурс таблицы строк.** У приложения единственный ресурс строковой таблицы с искусственным делением на сегменты, оставшимся с эпохи 16-разрядных программ. В окне Resource View дважды щелкните значок String Table — откроется редактор строк. Затем щелкните пустой элемент в конце списка и добавьте две строки:

Идентификатор строки	Текст (Caption)
ID_INDICATOR_LEFT	LEFT
ID_INDICATOR_RIGHT	RIGHT

По завершении операций таблица должна выглядеть так:



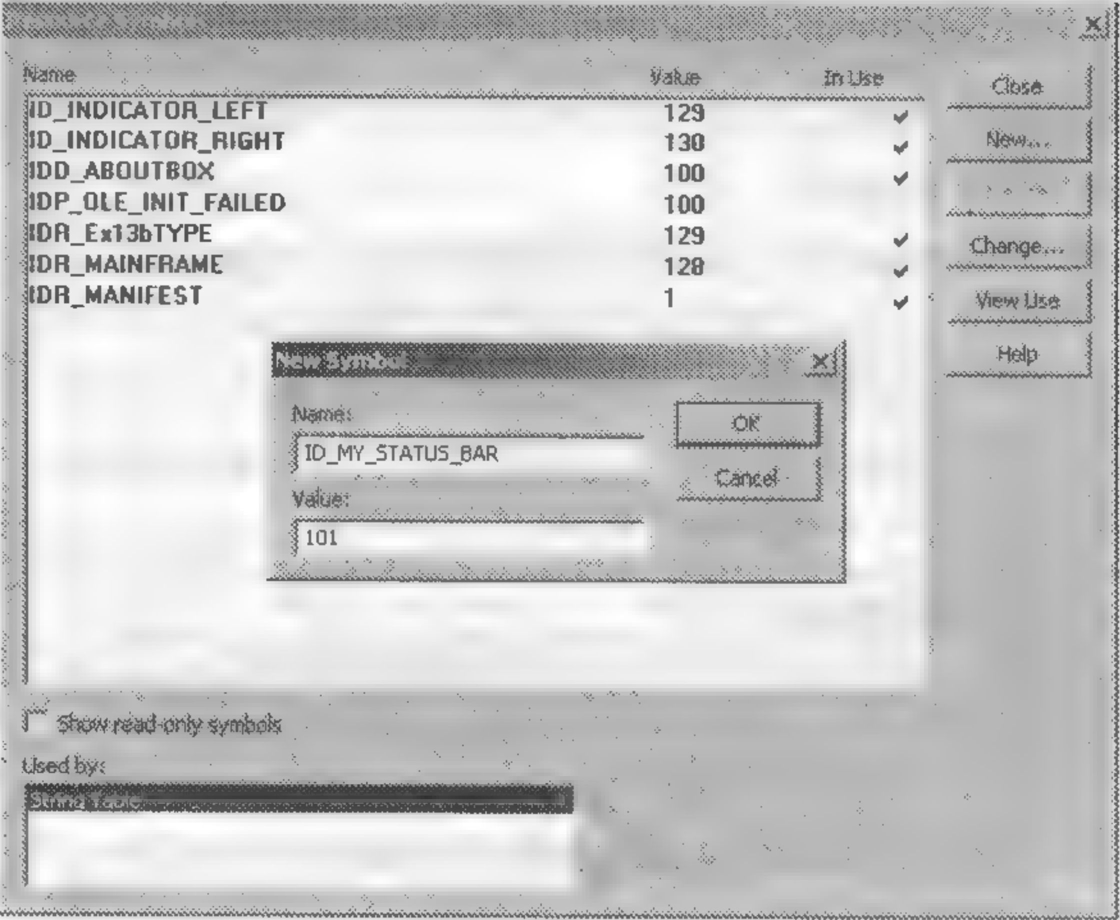
3. **Отредактируйте символы в ресурсах приложения.** Выберите в меню Edit команду Resource Symbols. Щелкните кнопку New и задайте для строки состояния новый идентификатор *ID_MY_STATUS_BAR*, а его значение оставьте по умолчанию (см. рисунок на следующей странице).
4. **В окне Properties утилиты Class View добавьте в класс *CMainFrame* обработчики команд, содержащихся в меню View.** Выберите класс *CMainFrame* в окне Class View, в окне Properties щелкните кнопку Events и добавьте обработчики командных сообщений:

Идентификатор объекта	Сообщение	Имя функции-члена
ID_VIEW_STATUS_BAR	*COMMAND*	*OnViewStatusBar*
ID_VIEW_STATUS_BAR	*UPDATE_COMMAND_UI*	*OnUpdateViewStatusBar*

5. **Вставьте в MainFrm.h прототипы функций.** Вам придется добавить эти прототипы обработчиков сообщений *CMainFrame* вручную, так как Visual Studio не распознает связанные с ними идентификаторы командных сообщений.

```
afx_msg void OnUpdateLeft(CCmdUI* pCmdUI);
afx_msg void OnUpdateRight(CCmdUI* pCmdUI);
```

Кроме того, объявите *m_wndToolBar* открытой, а не защищенной переменной.



6. **Отредактируйте файл MainFrm.cpp.** Замените старое содержимое массива *indicators* новым (оно выделено):

```
static UINT indicators[] =
{
   ID_SEPARATOR, // первая секция строки сообщений
   ID_SEPARATOR, // вторая секция строки сообщений
   ID_INDICATOR_LEFT,
   ID_INDICATOR_RIGHT,
};
```

Далее отредактируйте функцию-член *OnCreate*. Замените оператор:

```
if (!m_wndStatusBar.Create(this) ||
   !m_wndStatusBar.SetIndicators(indicators,
     sizeof(indicators)/sizeof(UINT)))
{
   TRACE0("Failed to create status bar\n");
   return -1;      // fail to create
}
```

на:

```
if (!m_wndStatusBar.Create(this,
      WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, ID_MY_STATUS_BAR) ||
   !m_wndStatusBar.SetIndicators(indicators,
     sizeof(indicators)/sizeof(UINT)))
{
   TRACE0("Failed to create status bar\n");
   return -1;      // fail to create
}
```

В модифицированном вызове *Create* вместо *AFX_IDW_STATUS_BAR* (объект строки состояния, формируемый каркасом приложений) используется наш идентификатор строки состояния *ID_MY_STATUS_BAR*.

Теперь добавьте следующие элементы таблицы сообщений для класса *CMainFrame*. Visual Studio не способна сделать это, так как не распознает идентификаторы из строковой таблицы в качестве идентификаторов объектов:

```
ON_UPDATE_COMMAND_UI(ID_INDICATOR_LEFT, OnUpdateLeft)
ON_UPDATE_COMMAND_UI(ID_INDICATOR_RIGHT, OnUpdateRight)
```

Затем вставьте функции-члены класса *CMainFrame*, отвечающие за обновление индикаторов:

```
void CMainFrame::OnUpdateLeft(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(::GetKeyState(VK_LBUTTON) < 0);
}
void CMainFrame::OnUpdateRight(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(::GetKeyState(VK_RBUTTON) < 0);
}
```

Заметьте: левой и правой кнопкам мыши, как и клавишам на клавиатуре, соответствуют коды виртуальных клавиш, поэтому при определении состояния кнопок можно обойтись без сообщений от мыши.

И, наконец, отредактируйте следующие функции для меню View в файле MainFrm.cpp:

```
void CMainFrame::OnViewStatusBar()
{
    m_wndStatusBar.ShowWindow((m_wndStatusBar.GetStyle() & WS_VISIBLE) == 0);
    RecalcLayout();
}
void CMainFrame::OnUpdateViewStatusBar(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck((m_wndStatusBar.GetStyle() & WS_VISIBLE) != 0);
}
```

Эти функции обеспечивают надлежащую связь команды Status Bar меню View с новой строкой состояния.

7. **Отредактируйте функцию *OnDraw* в Ex13bView.cpp.** Эта функция выводит сообщение в окне представления. Добавьте выделенный код:

```
void CEx13bView::OnDraw(CDC* pDC)
{
    CEx13bDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    pDC->TextOut(0, 0,
        "Watch the status bar while you move and click the mouse.");
}
```

8. **Добавьте в класс *CEx13bView* обработчик для WM_MOUSEMOVE.** Выберите класс *CEx13bView* в окне Class View, в окне Properties щелкните кнопку Messages, создайте функцию *OnMouseMove* и отредактируйте ее, как показано ниже. Эта функция получает указатель на объект строки состояния и вызывает *SetPaneText* для обновления первой и второй секции строки сообщений.

```
void CEx13bView::OnMouseMove(UINT nFlags, CPoint point)
{
    CString str;
    CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
    CStatusBar* pStatus = &pFrame->m_wndStatusBar;
    if (pStatus) {
        str.Format("x = %d", point.x);
        pStatus->SetPaneText(0, str);
        str.Format("y = %d", point.y);
        pStatus->SetPaneText(1, str);
    }
}
```

И, наконец, в начале файла Ex13bView.cpp вставьте директиву:

```
#include "MainFrm.h"
```

9. **Соберите и протестируйте приложение Ex13b.** Подвигайте мышь и убедитесь, что две левые секции в строке состояния точно отражают позицию курсора. Попробуйте нажать на левую и правую кнопки мыши. Удается ли включить/отключить показ строки состояния командами из меню View?

Примечание Чтобы у левой (нулевой) секции сообщений было обрамление, как и у остальных секций, а также чтобы строка состояния изменяла свой размер в соответствии с содержимым, включите в функцию *CMainFrame::OnCreate* (после вызова функции *Create* для строки состояния) две строки:

```
m_wndStatusBar.SetPaneInfo(0, 0, 0, 50);
m_wndStatusBar.SetPaneInfo(1, 0, SBPS_STRETCH, 50);
```

Они изменяют ширину первых двух секций (по сравнению с шириной по умолчанию в четверть экрана) и делают вторую секцию «растягивающейся».

Панель инструментов Rebar

Как вы знаете из главы 8, в Visual C++ есть компоненты, которые «пришли» из Internet Explorer. Речь идет о стандартных элементах управления, в частности о новой панели инструментов *rebar*. Скорее всего вы уже знакомы с ней, если хоть раз работали с Internet Explorer. От стандартной панели инструментов ее отличает MFC наличие *захватов* (gripper), позволяющих пользователю плавно изменять ее положение по горизонтали и по вертикали; положение стандартной панели меняется в результате стыковки при перетаскивании. Rebar-панели позволяют реализовать гораздо больше типов элементов управления (например, выпадающие меню), чем это позволяет сделать класс *CToolBar*.

Внутренняя структура rebar-панели

На рис. 13-5 показана структура rebar-панели и названия ее составных частей. Каждая внутренняя панель инструментов в rebar-панели называется *полосой* (band). Краешек с чертой, за которую пользователь перетаскивает полосу, называется *захватом* (gripper). Полоса может иметь *метку* (label).



Рис. 13-5. *Составные части rebar-панели*

В MFC есть два класса, облегчающие работу с rebar-панелями.

- ***CReBar*** — высокоуровневый класс абстрагирования, поддерживающий добавление объектов классов *CToolBar* и *CDialogBar* в rebar-панели в качестве полос. Кроме того, *CReBar* управляет (например, посредством уведомлений) взаимодействием между лежащим в его основе элементом управления и каркасом MFC-приложения.
- ***CReBarCtrl*** — низкоуровневый класс-оболочка элемента управления ReBar. В этом классе есть многочисленные элементы для создания и манипулирования rebar-панелями, но он не так удобен, как *CReBar*.

Большинство MFC-приложений работают с *CReBar*; чтобы получить доступ к низкоуровневым возможностям, вызывают функцию-член *GetReBarCtrl*, которая возвращает указатель на *CReBarCtrl*.

Пример Ex13c: rebar-панели

Познакомимся с возможностями rebar-панели на конкретном примере. Мы создадим SDI-приложение с rebar-панелью с двумя полосами: уже знакомой нам панелью инструментов и диалоговой панелью. На рис. 13-6 показано приложение Ex13c в работе.

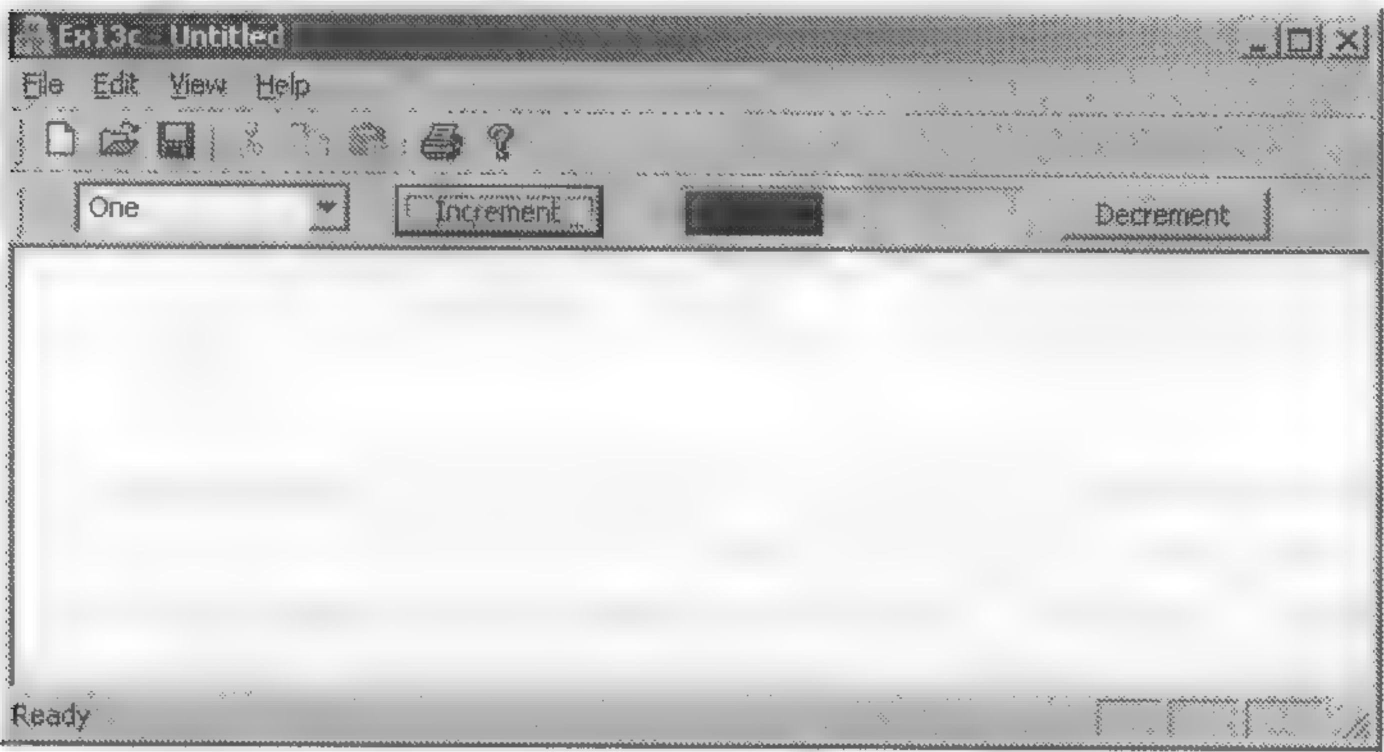
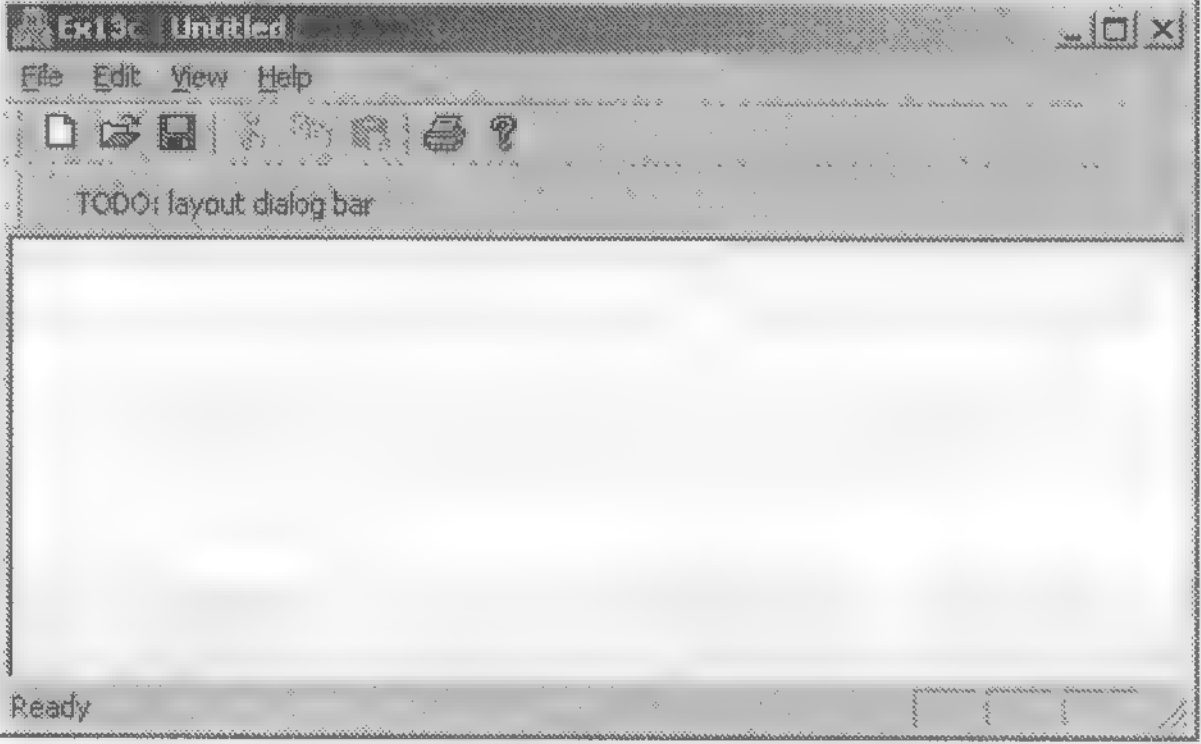


Рис. 13-6. *Программа Ex13c с rebar-панелью в работе*

1. **Средствами MFC Application Wizard создайте проект Ex13c.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC Application и в качестве имени проекта — Ex13c. На странице Application Type мастера установите переключатель в положение Single document, на странице User Interface Features установите флажок Browser Style, а переключатель Toolbars — в положение Standard Docking.
2. **Скомпилируйте и запустите приложение.** Запустив приложение, вы увидите, что MFC Application Wizard автоматически создал rebar-панель с двумя полосами. Одна содержит стандартную панель инструментов, а вторая — текст «TODO: layout dialog bar» («Доделать: разместить диалоговую панель»):



Теперь, открыв файл MainFrm.h, вы увидите приведенный ниже код, в котором объявляется элемент данных *m_wndReBar* класса *CReBar*.

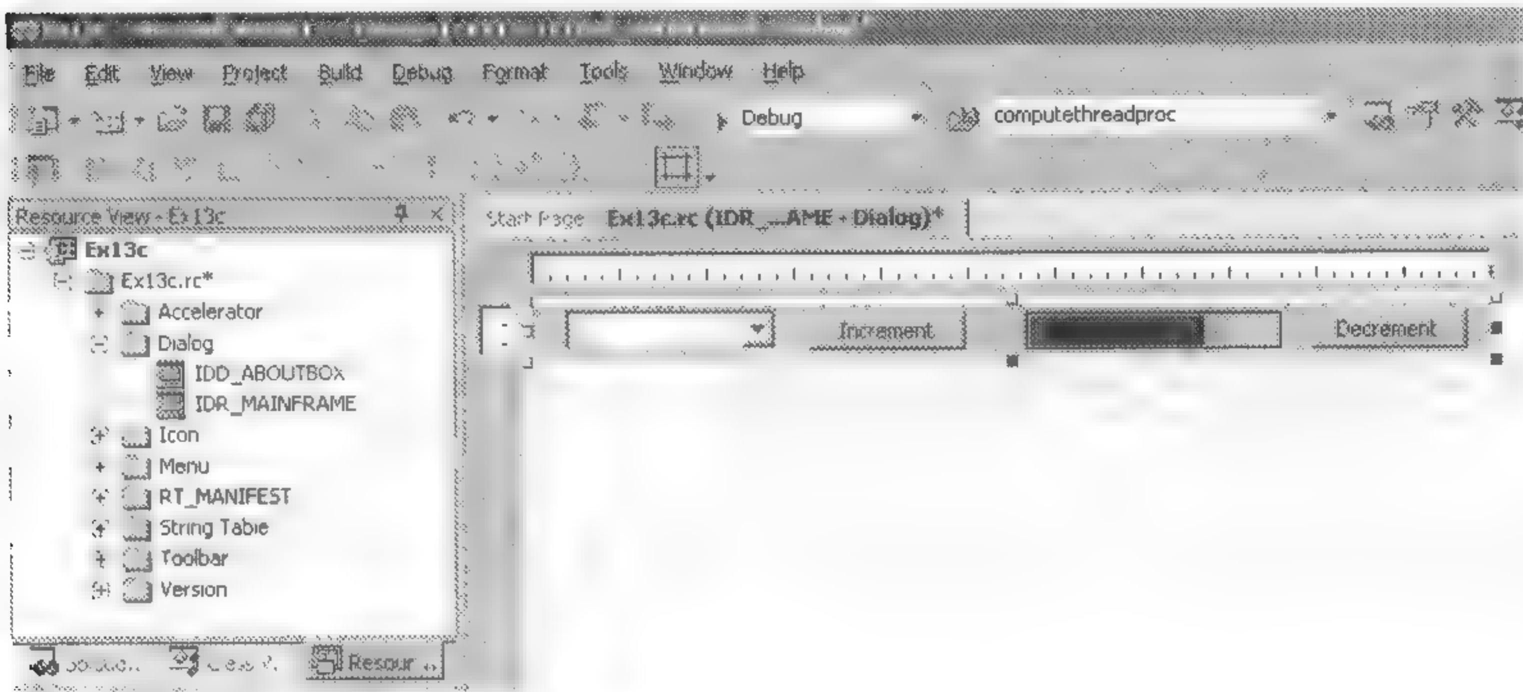
```
protected:  // control bar embedded members
   CStatusBar  m_wndStatusBar;
   CToolBar  m_wndToolBar;
   CReBar   m_wndReBar;
   CDialogBar  m_wndDlgBar;
```

А в файле MainFrame.cpp можно увидеть код, который вставляет панель инструментов и диалоговую панель в объект *CReBar*.

```
if (!m_wndReBar.Create(this) ||
      !m_wrdReBar.AddBar(&m_wndToolBar) ||
      !m_wndReBar.AddBar(&m_wndDlgBar))
   {
      TRACE0("Failed to create rebar\n");
      return -1;     // fail to create
   }
```

3. **Измените диалоговую панель.** В окне Resource View в узле Dialog найдите диалоговый ресурс с идентификатором *IDR_MAINFRAME*. Открыв его, вы увидите диалоговую панель с текстом «TODO: layout dialog bar» («Доделать: разместить диалоговую панель»). Последуем этому дружескому совету и разместим здесь несколько элементов управления. Прежде всего удалите статический элемент с текстом «TODO...». Затем поместите в диалоговую панель поле со

списком и в окне Properties введите в него (точнее, в свойство Data) несколько элементов данных: One;Two;Buckle;My;Shoe!;. Затем поместите в диалоговую панель кнопку и измените ее свойство Caption на *Increment*. А теперь добавьте на панель индикатор хода процесса с установленными по умолчанию значениями свойств. Наконец, добавьте еще одну кнопку с названием *Decrement*. После окончания работы диалоговая панель должна выглядеть так.



4. **Отредактируйте файл MainFrm.h.** Visual Studio не «знает», как связать элементы управления панель с обработчиками в классе *CMainFrame*. Их придется добавлять вручную. Откройте файл MainFrm.h и добавьте прототипы в *CMainFrame*.

```
afx_msg void OnButton1();
afx_msg void OnButton2();
```

5. **Отредактируйте файл MainFrm.cpp.** Откройте файл MainFrm.cpp и создайте в карте сообщений записи для кнопок *Button1* и *Button2*.

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_BN_CLICKED(IDC_BUTTON1, OnButton1)
    ON_BN_CLICKED(IDC_BUTTON2, OnButton2)
END_MESSAGE_MAP()
```

Добавьте методы *OnButton1* и *OnButton2* в CMainFrm.cpp:

```
void CMainFrame::OnButton1()
{
    CProgressCtrl * pProgress =
      (CProgressCtrl*)m_wndDlgBar.GetDlgItem(IDC_PROGRESS1);
    pProgress->StepIt();
}

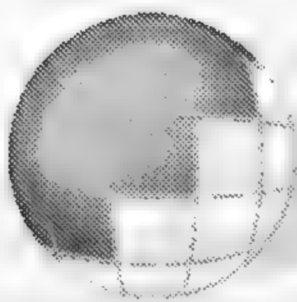
void CMainFrame::OnButton2()
{
    CProgressCtrl * pProgress =
      (CProgressCtrl*)m_wndDlgBar.GetDlgItem(IDC_PROGRESS1);
    int nCurrentPos = pProgress->GetPos();
    pProgress->SetPos(nCurrentPos-10);
}
```

Обработчик *OnButton1* получает указатель на индикатор хода процесса и вызывает *StepIt* для увеличения показаний элемента управления. *OnButton2* уменьшает значение элемента управления на 10.

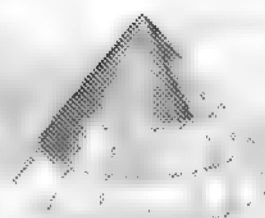
6. **Скомпилируйте и протестируйте приложение.** Теперь можно скомпилировать и запустить Ex13c, чтобы увидеть отредактированную rebar-панель в действии. Кнопка Increment увеличивает индикатор продвижения, а Decrement — уменьшает.

ГЛАВА

14







Повторно используемый базовый класс окна-рамки

Язык C++ позволяет задействовать «конструктор» из программных блоков, которые можно запросто брать «с полки» и вставлять в новые приложения. Прекрасный пример этого — классы библиотеки MFC. В этой главе мы рассмотрим, как создать свой повторно используемый базовый класс на основе библиотеки MFC.

Работая над таким классом, вы узнаете много нового о Windows и библиотеке MFC. В частности, вы увидите, как каркас приложений обеспечивает доступ к реестру Windows, разберетесь в механизме работы класса *CFrameWnd* и расширите свое представление о статических переменных класса и о классе *CString*.

Почему так трудно создавать повторно используемые базовые классы

В обычном приложении пишутся программные компоненты, предназначенные для решения конкретной задачи, и единственный критерий «истины» — требования к проекту. Однако, создавая повторно используемый базовый класс, нужно учитывать будущие потребности — как свои, так и других программистов. Класс должен быть не только универсальным и законченным, но эффективным и простым в работе.

Разрабатывая пример для этой главы, авторы воочию убедились, насколько трудно создавать повторно используемое ПО. Хотелось написать класс окна-рамки, который «запоминал» бы размеры и позицию своего окна. Вскоре выяснилось, что существующие Windows-программы запоминают еще и состояние своего окна: свернуто оно в значок или развернуто во весь экран. Вдобавок обнаружился какой-то странный тип окон: и свернутых в значок, и развернутых в полный экран одновременно. Еще надо было позаботиться о панели инструментов и строке со-

стояния и, кроме того, добиться, чтобы класс работал в DLL. Короче, написать класс окна-рамки, который делал бы все, что захочется программисту, оказалось страшно сложно.

При промышленной разработке ПО повторно используемые базовые классы зачастую не вписываются в нормальный цикл создания программ. Класс, подготовленный для одного проекта, берут и подгоняют для другого. При этом всегда испытываешь искушение «вырезать и вставить» существующие классы, не задаваясь вопросом: «А нельзя ли что-либо выделить в базовый класс?» Но если вы решили заниматься программированием всерьез, собственная библиотека настоящих повторно используемых программных компонентов будет как нельзя кстати.

Класс *CPersistentFrame*

В этой главе вы поработаете с классом *CPersistentFrame*, производным от *CFrameWnd*. Класс *CPersistentFrame* поддерживает *постоянную* (persistent) окно-рамку SDI-приложения, запоминающее следующие характеристики:

- размер окна;
- его положение;
- развернутое состояние;
- свернутое состояние;
- расположение и состояние видимости панели инструментов и строки состояния.

Завершая работу, приложение, в котором применяется класс *CPersistentFrame*, сохраняет указанную информацию в реестре. При следующем запуске приложение считывает эту информацию из реестра и восстанавливает состояние окна-рамки, существовавшее на момент завершения программы.

Этот класс окна-рамки пригоден для любого SDI-приложения, в том числе для примеров из этой книги. Все, что нужно сделать, — заменить *CFrameWnd* в производном классе окна-рамки приложения на *CPersistentFrame*.

Класс *CFrameWnd* и функция-член *ActivateFrame*

Почему в качестве базового класса для постоянного окна выбран *CFrameWnd*? Почему бы вместо этого не создать класс *постоянного вида* (persistent view)? В SDI-приложении, написанном с использованием MFC, окно-рамка всегда является родительским по отношению к окну класса «вид». Сначала создается окно-рамка, и только потом формируются его дочерние окна: панель инструментов, строка состояния и окно представления (объект «вид»). Каркас приложений следит за соответствующим изменением размеров дочерних окон при масштабировании окна-рамки, поэтому изменять размер окна представления после создания окна-рамки нет смысла.

Функция-член *CFrameWnd::ActivateFrame* — ключ к управлению размером окна-рамки. Каркас приложений вызывает эту виртуальную функцию, объявленную в классе *CFrameWnd*, при создании основного окна-рамки SDI-приложения (и по командам New или Open из меню File). Задача каркаса — вызвать функцию *CWnd::ShowWindow* с параметром *nCmdShow*. *ShowWindow* делает видимым окно-рамку вместе с меню, окном представления, панелью инструментов и строкой состояния. Параметр *nCmdShow* определяет, развернуто или свернуто окно.

Если переопределить функцию *ActivateFrame* в производном классе, можно изменять значение *nCmdShow* перед передачей его в функцию *CFrameWnd::ActivateFrame*. Кроме того, можно вызвать функцию *CWnd::SetWindowPlacement*, которая устанавливает размер и позицию окна-рамки и позволяет указать, должны ли в нем присутствовать панель инструментов и строка состояния. Поскольку все изменения выполняются до того, как окно-рамка становится видимой, на экране не возникнет неприятного мелькания.

Нужно позаботиться и о том, чтобы не инициализировать повторно положение и размер окна-рамки после каждой команды File New и File Open. Для этого служит переменная-член — признак первого вызова, проверка которого гарантирует, что функция *CPersistentFrame::ActivateFrame* вызывается только при запуске приложения.

Функция-член *PreCreateWindow*

PreCreateWindow, объявленная в *CWnd*, — еще одна виртуальная функция, которую можно переопределить, чтобы изменить характеристики окна до его появления на экране. Каркас приложений вызывает ее перед *ActivateFrame*. Мастер MFC Application Wizard *всегда* генерирует в ваших классах «вид» и «окно-рамка» переопределенную функцию *PreCreateWindow*.

В качестве параметра этой функции передается структура *CREATESTRUCT*, в которой есть две переменных-члена: *style* и *dwExStyle*. Вы можете изменить их перед передачей структуры в функцию *PreCreateWindow* из базового класса. Флажок *style* определяет, есть ли у окна границы, полосы прокрутки, кнопка сворачивания окна и т. п. Флажок *dwExStyle* управляет другими характеристиками, в том числе расположением поверх остальных окон. Подробнее об этом см. в разделах Window Styles и Extended Window Styles интерактивной документации по библиотеке MFC.

Элемент *lpszClass* структуры *CREATESTRUCT* позволяет изменять кисть фона, курсор или значок окна. Изменять курсор или фон для окна-рамки бессмысленно: его клиентская область закрыта окном представления. Если вы, к примеру, захотите создать окно представления с жутким красным фоном, переопределите *PreCreateWindow* для класса «вид»:

```
BOOL CMyView::PreCreateWindow(CREATESTRUCT& cs)
{
   if (!CView::PreCreateWindow(cs)) {
      return FALSE;
   }
   cs.lpszClass =
         AfxRegisterWndClass(CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
                 AfxGetApp()->LoadCursor(IDC_MYCURSOR),
                 ::CreateSolidBrush(RGB(255, 0, 0)));
   if (cs.lpszClass != NULL) {
      return TRUE;
   }
   else {
      return FALSE;
   }
}
```

Если переопределить *PreCreateWindow* в классе постоянного окна-рамки, окна всех производных классов будут обладать свойствами, заложенными в базовый класс. Конечно, в производных классах функцию *PreCreateWindow* можно снова переопределить, но тогда надо продумать взаимодействие этих двух функций.

Реестр Windows

Если вы работали с приложениями для 16-разрядной Windows, то, вероятно, знакомы с INI-файлами. Их можно применять и в Win32-приложениях, но Microsoft рекомендует вместо этого использовать реестр — набор системных файлов, контролируемых Windows, куда ОС и отдельные программы могут помещать информацию для долговременного хранения. Реестр организован как иерархическая база данных, доступ к строковым и числовым данным которой осуществляется по составным ключам.

Допустим, текстовый процессор TEXTPROC «хочет» хранить в реестре тип и размер последнего использованного шрифта. Пусть имя программы значится в корневом разделе (это, конечно, упрощение), а приложение поддерживает два уровня иерархии в этом разделе. Получится структура вроде этой:

```
TEXTPROC
    Text formatting
        Font = Times Roman
        Points = 10
```

Для доступа к реестру MFC-библиотека предоставляет 4 функции-члена класса *CWinApp*, сохранившиеся еще со времен INI-файлов. MFC Application Wizard генерирует вызов *CWinApp::SetRegistryKey* в функции *InitInstance* программы:

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

Если этот вызов удалить, приложение будет использовать не реестр Windows, а INI-файлы в каталоге Windows. Строковый параметр функции *SetRegistryKey* определяет верхний уровень иерархии, а перечисленные ниже «реестровые» функции определяют два нижних уровня: с именем подраздела и с именем параметра.

- *GetProfileInt*;
- *WriteProfileInt*;
- *GetProfileString*;
- *WriteProfileString*.

Эти функции трактуют данные в реестре либо как объекты *CString*, либо как целые числа без знака. Если нужно сохранять в реестре значения с плавающей запятой, придется использовать строковые функции и заняться преобразованиями. Всем функциям передаются имена подраздела и параметра реестра. В предыдущем примере TEXTPROC у подраздела назывался Text formatting, а параметры — Font и Points.

Чтобы вызвать функции доступа к реестру, нужен указатель на объект-приложение. Его позволяет получить глобальная функция *AfxGetApp*. В предыдущем примере установить параметры Font и Points может такой код:

```
AfxGetApp()->WriteProfileString("Text formatting", "Font", "Times Roman");
AfxGetApp()->WriteProfileInt("Text formatting", "Points", 10);
```

Unicode

Для кодирования всех символов европейских языков (в том числе с диакритическими знаками) хватает 8 разрядов, большинство азиатских языков требует 16. Во многих программах применяется стандарт, предусматривающий 2-байтовое представление символов (double-byte character set, DBCS). В таком наборе одни символы кодируются 8, а остальные — 16 разрядами в зависимости от значения первых 8 разрядов. На смену 2-байтового стандарта приходит Unicode, где все символы кодируются 16 разрядами. Для отдельных языков не выделяются отдельные наборы символов: так, если символ используется в японском и китайском языках, то он входит лишь раз в набор символов Unicode.

В исходном коде MFC и коде, сгенерированном MFC Application Wizard, можно увидеть типы *TCHAR*, *LPTSTR* и *LPCTSTR*, а также строковые константы вида *_T(«строка»)*. Перед нами *макросы* Unicode. Если собирать проект без определения символа препроцессора *_UNICODE*, компилятор сгенерирует код для обычных 8-разрядных символов ANSI (CHAR) и указателей на массивы 8-разрядных символов (*LPSTR*, *LPCSTR*). Если же определить *_UNICODE*, компилятор сгенерирует код для 16-разрядных символов Unicode (*WCHAR*), соответствующих указателей (*LPWSTR*, *LPCWSTR*) и строковых констант (*L«строка в Unicode-представлении»*).

Символ препроцессора *_UNICODE* определяет также, какие функции вызывает приложение, так как многие функции Win32 существуют в двух версиях. Когда программа вызывает, например, *CreateWindowEx*, компилятор генерирует код для вызова либо *CreateWindowExA* (с ANSI-параметрами), либо *CreateWindowExW* (с Unicode-параметрами). В Windows NT, 2000, XP, где используется Unicode, *CreateWindowExW* передает все параметры системе напрямую, тогда как *CreateWindowExA* преобразует строковые (ANSI) и символьные параметры в Unicode. А вот в Windows 95, 98, где используется ANSI, *CreateWindowExW* представляет собой заглушку, которая возвращает ошибку, а *CreateWindowExA* передает параметры системе напрямую.

Если вы хотите создать Unicode-приложение, его следует создавать для Windows NT/2000/XP и везде использовать макросы. Можно создавать Unicode-приложения и для Windows 95/98, но тогда придется проделать лишнюю работу, явно вызывая A-версии функций Win32. Как показано в главах 24–30, в вызовах COM всегда используются Unicode-символы. Функции Win32 для преобразования между ANSI и Unicode существуют, но класс *CString* позволяет для этого прибегнуть к помощи Unicode-конструктора и функции-члена *AllocSysString*.

Простоты ради во всех примерах этой книги применяется только ANSI. Код, создаваемый мастером MFC Application Wizard, содержит макросы Unicode, но в коде, написанном авторами, используются строковые константы 8-разрядных символов и типы *char*, *char** и *const char**.

В примере Ex14a вы поработаете с реестром, научитесь просматривать и редактировать его с помощью программы Regedit.

Примечание Каркас приложений сохраняет список последних открытых файлов в подразделе Recent File List.

Класс *CString*

MFC-класс *CString* значительно расширяет язык C++. У класса *CString* много полезных операторов и функций-членов, но, видимо, главное его достоинство — способность динамически выделять память. Это избавляет вас от забот о размере строки. Вот несколько типичных примеров использования объектов *CString*:

```
CString strFirstName("Elvis");
CString strLastName("Presley");
CString strTruth = strFirstName + " " + strLastName; // конкатенация
strTruth += " is alive";
ASSERT(strTruth == "Elvis Presley is alive");
ASSERT(strTruth.Left(5) == strFirstName);
ASSERT(strTruth[2] == 'v'); // операция индексирования
```

В идеальном мире программы на C++ всегда использовали бы объекты *CString* и никогда — обычные массивы символов с нулем в конце. Увы, многие функции стандартной библиотеки по-прежнему полагаются на эти массивы, поэтому программы должны уметь работать с обоими представлениями строк. К счастью, в классе *CString* есть оператор *const char* ()*, преобразующий объект *CString* в указатель на символы. Многие функции библиотеки MFC требуют параметров типа *const char**. Возьмем глобальную функцию *AfxMessageBox*. Вот один из ее прототипов:

```
int AFXAPI AfxMessageBox(LPCTSTR lpszText, UINT nType = MB_OK,
                         UINT nIDHelp = 0);
```

(Заметьте: *LPCTSTR* — это не указатель на объект *CString*, а совместимый с Unicode заменитель для *const char**.)

AfxMessageBox можно вызвать так:

```
char szMessageText[] = "Unknown error";
AfxMessageBox(szMessageText);
```

или так:

```
CString strMessageText("Unknown error");
AfxMessageBox(strMessageText);
```

Теперь предположим, что надо сгенерировать строку в соответствии с определенным форматом. Эту задачу решает функция *CString::Format*:

```
int nError = 23;
CString strMessageText:
strMessageText.Format("Error number %d", nError);
AfxMessageBox(strMessageText);
```

Примечание Допустим, вам понадобился прямой доступ к символам в объекте *CString*. Если вы напишете примерно такой код:

```
CString strTest("test");
strncpy(strTest, "T", 1);
```

компилятор сообщит об ошибке, так как первый параметр *strncpy* объявлен как *char**, а не *const char**. Функция *CString::GetBuffer* фиксирует заданный размер буфера и возвращает *char**. Чтобы потом вновь сделать строку динамической, вызовите функцию-член *ReleaseBuffer*. Вот как правильно заменить строчную букву *t* на прописную:

```
CString strTest("test");
strncpy(strTest.GetBuffer(5), "T", 1);
strText.ReleaseBuffer();
ASSERT(strTest == "Test");
```

Оператор *const char** преобразует объект *CString* в указатель на константу; но как быть с обратным преобразованием? У класса *CString* есть конструктор, преобразующий указатель на символьную константу в объект *CString*, и, кроме того, набор переопределенных операторов для таких указателей. Вот почему допустимы такие выражения, как:

```
truth += " is alive";
```

Специальный конструктор работает с функциями, принимающими ссылку на *CString*, например с *CDC::TextOut*. В следующем примере в стеке вызывающей программы создается временный объект *CString*, а его адрес передается в *TextOut*:

```
pDC->TextOut(0, 0, "Hello, world!");
```

Если надо самостоятельно подсчитать число символов, эффективнее использовать переопределенную версию *CDC::TextOut*:

```
pDC->TextOut(0, 0, "Hello, world!", 13);
```

При написании функции, принимающей строковый параметр следует соблюдать несколько правил.

- Если функция не меняет содержимое строки и вы собираетесь работать со стандартными библиотечными функциями, такими как *strcpy*, используйте параметр *const char**.
- Если функция не меняет содержимое строки, но вы хотите вызывать в ней функции-члены *CString*, используйте параметр *const CString&*.
- Если функция меняет содержимое строки, используйте параметр *CString&*.

Полностью развернутое окно

Окно в Windows можно развернуть либо через системное меню, либо щелкнув кнопку в его правом верхнем углу. Аналогично можно восстановить окно (вернуть его к исходному размеру из развернутого состояния). То есть развернутое окно «помнит» свои исходные размеры и положение.

Экранные координаты окна возвращает функция *GetWindowRect* класса *CWnd*. Если окно развернуто, она сообщает размер экрана, а не координаты окна в неразвернутом состоянии. Чтобы класс постоянного окна-рамки работал с развер-

нутым окном, он должен «знать» его координаты в обычном состоянии. Эти координаты — вместе с флажками, которые указывают, развернуто или свернуто окно в данный момент, — возвращает функция *CWnd::GetWindowPlacement*.

Парная ей функция *SetWindowPlacement* позволяет задать положение и размеры окна как в развернутом, так и в свернутом состояниях. Чтобы вычислить положение левого верхнего угла развернутого окна, надо учесть размер границ окна, который позволяет выяснить Win32-функция *GetSystemMetrics*. Код функции *CPersistentFrame::ActivateFrame*, где используется *SetWindowPlacement*, приведен далее в листинге файла Persist.cpp.

Состояние панелей элементов управления и реестр

Для сохранения в реестре и последующего восстановления состояния панелей элементов управления в MFC-библиотеке служат две функции-члена класса *CFrameWnd*: *LoadBarState* и *SaveBarState*. Обе применимы и к строке состояния, и к стыкуемым панелям инструментов, но не обрабатывают положение плавающих панелей инструментов.

Статические переменные-члены

Класс *CPersistentFrame* хранит имена своих разделов реестра в переменных-членах типа массивы *static const char*. Какие есть еще варианты? Строковые ресурсы не годятся, так как строки надо определять в самом классе. (Однако использование строковых ресурсов имеет смысл, если *CPersistentFrame* встроен в DLL.) Глобальные переменные обычно не рекомендуются, так как нарушают инкапсуляцию. Применять статические объекты *CString* бессмысленно, поскольку тогда при запуске программы строки пришлось бы копировать в динамическую память.

Еще один очевидный вариант — обычные переменные-члены. Но статические лучше, так как, будучи константами, они выделяются в секцию данных «только для чтения» и могут предоставляться сразу нескольким работающим копиям одной программы. Когда класс *CPersistentFrame* — часть DLL-модуля, массивы символов можно спроецировать на все процессы, использующие DLL. По сути статические переменные-члены — это глобальные переменные, область видимости которых ограничена соответствующим классом, что исключает конфликт имен.

Оконный прямоугольник по умолчанию

Вы уже определяли прямоугольники в аппаратных или логических координатах. Объект *CRect*, создаваемый оператором:

```
CRect rect(CW_USEDEFAULT, CW_USEDEFAULT, 0, 0);
```

имеет особый смысл. Создавая новое окно с таким прямоугольником, Windows смещает его по диагонали вправо и вниз относительно последнего открытого окна. При этом правый и нижний края окна всегда находятся в пределах экрана.

Такой особый прямоугольник содержится в статической переменной-члене *rectDefault* класса *CFrameWnd*, которая конструируется именно с использованием констант *CW_USEDEFAULT*. Класс *CPersistentFrame* объявляет как статическую переменную-член собственный оконный прямоугольник по умолчанию, *rectDefault*,

с фиксированным размером и положением, скрывая тем самым переменную базового класса.

Пример Ex14a: класс постоянного окна-рамки

Программа Ex14a иллюстрирует применение класса постоянного окна-рамки (*CPersistentFrame*). На рис. 14-1 показано содержимое файлов Persist.h и Persist.cpp из проекта Ex14a на компакт-диске. В этом примере мы вставим новый класс окна-рамки в SDI-приложение, сгенерированное MFC Application Wizard. Ex14a — это «ничего не делающая» программа, но класс постоянного окна-рамки можно легко перенести в другое SDI-приложение, которое делает что-то полезное.

Persist.h

```
[illegible]
```

Persist.cpp

```
[illegible]
```

см. след. стр.



[illegible]

см. след. стр.

```
[illegible]
```

Итак, создаем приложение Ex14a.

1. **Средствами MFC Application Wizard создайте проект Ex14a.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Advanced Features сбросьте флажок Printing and print preview. Остальные параметры оставьте без изменения.
2. **Измените MainFrm.h.** Вы должны изменить базовый класс для *CMainFrame*. Для этого просто замените строку:

```
class CMainFrame : public CFrameWnd
```

на:

```
class CMainFrame : public CPersistentFrame
```

Не забудьте добавить строку:

```
[illegible]
```

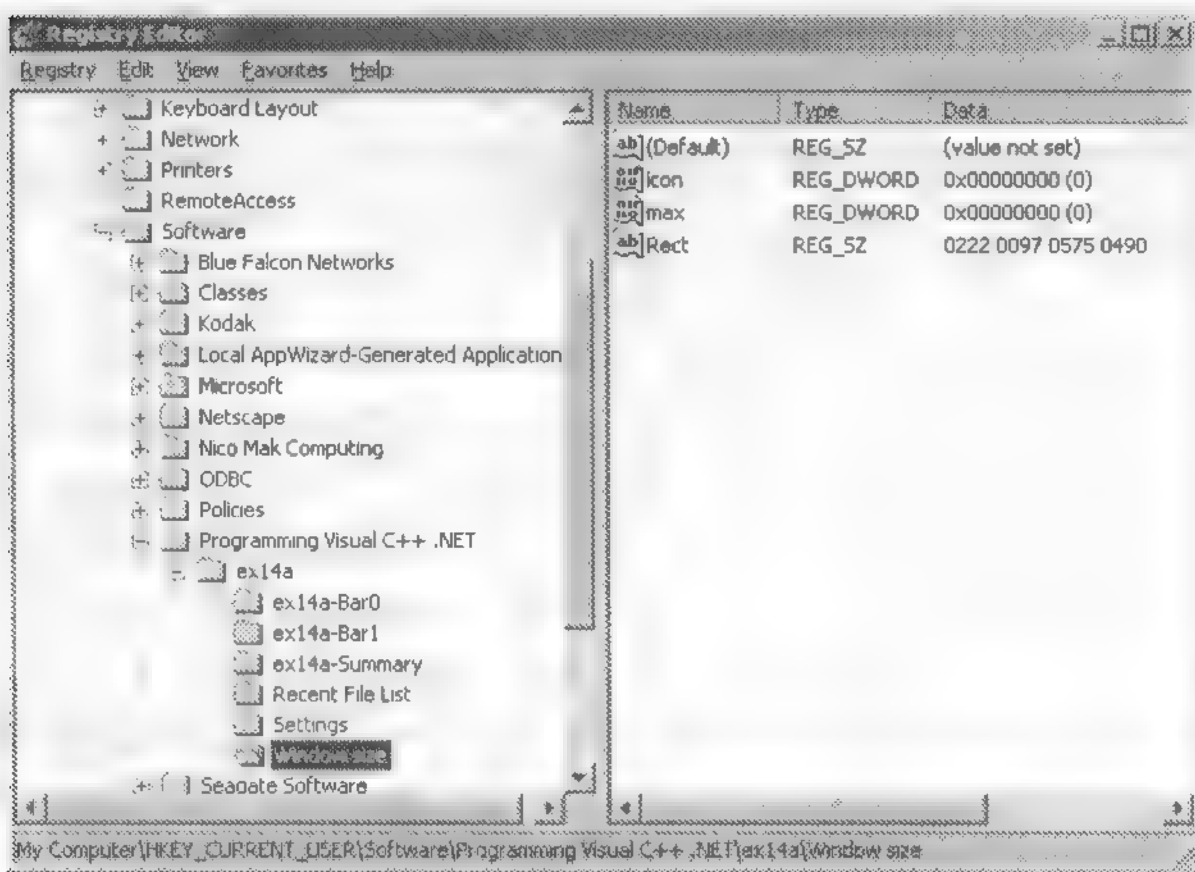
3. **Измените MainFrm.cpp.** Проведите глобальный поиск и замену всех вхождений *CFrameWnd* на *CPersistentFrame*.
4. **Отредактируйте Ex14a.cpp.** Замените строку:

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

на:

```
SetRegistryKey("Programming Visual C++ .NET");
```

5. **Добавьте файл Persist.cpp в проект.** Вы можете вручную набрать текст файлов Persist.h и Persist.cpp по листингам или скопировать их с компакт-диска. Но наличия этих файлов в каталоге \vcpp32\Ex14a недостаточно — вы должны добавить файл реализации в проект. В Visual C++ .NET из меню Project выберите команду Add Existing Item и в списке файлов — Persist.h и Persist.cpp.
6. **Соберите и протестируйте приложение Ex14a.** Измените размер и положение окна-рамки и закройте приложение. Перезапустите программу и проверьте, располагается ли окно на том же месте и сохранились ли его размеры. Поэкспериментируйте с разворачиванием и сворачиванием окна программы, затем отключите панель инструментов и строку состояния. Запомнило ли постоянное окно-рамка новые параметры?
7. **Просмотрите реестр Windows.** Запустите программу regedit.exe. Перейдите в раздел HKEY_CURRENT_USER\Software\Programming Visual C++ .NET\Ex14a. Вы должны увидеть там параметры, аналогичные следующим:



Обратите внимание на взаимосвязь раздела реестра и параметра «Programming Visual C++ .NET» в функции *SetRegistryKey*. Если в качестве параметра *SetRegistryKey* задать пустую строку, имя программы (в данном случае Ex14a) будет расположено прямо в разделе Software.

Постоянные рамки в MDI-приложениях

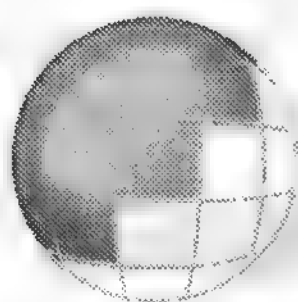
С MDI-приложениями мы начнем знакомиться в главе 16, но если вы используете эту книгу как справочник, вам может понадобиться технология работы с постоянной окном-рамкой в MDI-приложениях.

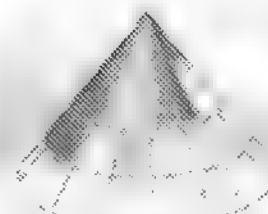
Класс *CPersistentFrame* в том виде, в каком представлен здесь, в MDI-приложении работать не будет, так как функцию *ShowWindow* основного окна-рамки MDI-приложения вызывает не виртуальная функция *ActivateFrame*, а сама функция-член *InitInstance* класса приложения. Чтобы управлять характеристиками основного окна-рамки MDI-приложения, добавьте в *InitInstance* необходимый код.

Однако функция *ActivateFrame* вызывается для объектов *CMDIChildWnd*. А это значит, что MDI-приложение способно запоминать размеры и положение своих дочерних окон. Эту информацию можно сохранить в INI-файле, но при этом нужно учесть наличие нескольких окон, а для этого придется модифицировать класс *CPersistentFrame*.

ГЛАВА

15





Документ и его представление

В данной главе мы наконец рассмотрим процесс взаимодействия документа и его представления. Первое впечатление об этом процессе вы получили в главе 12, когда речь шла о доставке сообщений объектам «вид» и «документ». А здесь вы увидите, как объект «документ» хранит данные, обрабатываемые программой, а объект «вид» представляет их пользователю. Вы также узнаете, как объекты «документ» и «вид» обмениваются данными при выполнении программы.

В обоих примерах этой главы для классов «вид» в качестве базового используется класс *CFormView*. В первом, предельно простом примере документ содержит единственный объект класса *CStudent*, представляющий одну запись о студенте. Окно представления отображает имя и общий балл студента и позволяет их изменять. Работая с классом *CStudent*, вы научитесь писать классы, отражающие объекты реального мира, и работать с функциями диагностического дампа библиотеки MFC.

Второй пример расширяет первый, вводя классы наборов указателей, в частности *CObList* и *CTypedPtrList*, что позволяет хранить в документе набор записей о студентах и обеспечить просмотр, вставку и удаление отдельных записей в окне представления.

Функции взаимодействия «документ-вид»

Вы знаете, что объект «документ» содержит данные, а объект «вид» представляет их на экране и позволяет редактировать. В SDI-приложении есть класс «документ», производный от *CDocument*, и один или несколько классов «вид», каждый из которых в конечном счете происходит от *CView*. Документ, окно представления и остальные элементы каркаса приложений взаимодействуют весьма тесно. Чтобы

в этом разобраться, надо познакомиться с пятью важными функциями-членами классов «документ» и «вид». Две из них — невиртуальные функции базового класса, вызываемые производными классами, остальные — виртуальные функции, часто переопределяемые в производных классах. Рассмотрим их по порядку.

Функция *CView::GetDocument*

С объектом «вид» связан единственный объект-документ. Функция *GetDocument* позволяет получать указатель на документ, соответствующий данному окну представления. Пусть объект «вид» получил сообщение о вводе пользователем новых данных в поле ввода. Он должен уведомить об этом документ, чтобы тот обновил свои внутренние данные. *GetDocument* возвращает указатель на документ; через этот указатель можно обращаться к функциям-членам или открытым переменным-членам класса «документ».

Примечание Функция *CDocument::GetNextView* позволяет перейти от документа к представлению, но так как у документа бывает несколько представлений, ее надо вызывать в цикле для каждого из них. Впрочем, прибегать к *GetNextView* придется нечасто — каркас приложений предоставляет более эффективный способ перебора представлений документа.

Генерируя класс, производный от *CView*, MFC Application Wizard создает специальные версии функции *GetDocument* — с поддержкой отладки и без — для безопасного приведения типов; она возвращает указатель не на *CDocument*, а на объект производного класса. Версия без поддержки отладки (содержится в заголовочном файле) выглядит примерно так:

```
inline CMyDoc* CMyView::GetDocument() const
   { return reinterpret_cast<CMyDoc*>(m_pDocument); }
```

Версия с поддержкой отладки (хранится в исходном файле представления и компилируется при включении отладки) имеет такой вид:

```
CMyDoc* CMyView::GetDocument() const    // версия без поддержки отладки
                                        //является встраиваемой (inline)
{
   ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyDoc)));
   return (CMyDoc*)m_pDocument;
}
```

Встретив в коде класса «вид» вызов *GetDocument*, компилятор вместо *CView::GetDocument*, возвращающей *CDocument**, использует *CMyView::GetDocument*, которая возвращает *CMyDocument**, поэтому вам не надо приводить возвращаемый указатель к производному классу документа. Без подобной вспомогательной функции компилятор вызывал бы функцию базового класса *GetDocument* и возвращал указатель на объект *CDocument*.

Следующий оператор всегда вызывает функцию базового класса *GetDocument* независимо от наличия указанной вспомогательной функции, так как функция *CView::GetDocument* не является виртуальной.

```
pView->GetDocument(); // pView объявлен как CView*
```

Функция *CDocument::UpdateAllViews*

Если содержание документа почему-либо изменилось, надо уведомить об этом все объекты «вид», чтобы те обновили представление данных. При вызове *UpdateAllViews* из функции-члена производного класса документа ее первый параметр *pSender* равен *NULL*. Если же она вызывается из функции-члена производного класса «вид», параметр *pSender* указывает на текущий объект «вид»:

```
GetDocument()->UpdateAllViews(this);
```

Значение параметра, отличное от *NULL*, позволяет каркасу приложений не уведомлять текущий объект «вид» — предполагается, что он уже обновился сам.

У этой функции есть необязательные параметры, через которые можно передавать объекту «вид» характерную для приложения информацию о том, какие части представления обновить. Это более «навороченный» способ применения функции.

Чтобы узнать, как именно уведомляется объект «вид» при вызове *UpdateAllViews*, познакомьтесь с функцией *OnUpdate*.

Функция *CView::OnUpdate*

Эта виртуальная функция вызывается каркасом приложений в ответ на вызов программой функции *CDocument::UpdateAllViews*. Конечно, вы вправе вызывать ее прямо в своем классе, производном от *CView*. Обычно *OnUpdate* производного класса обращается к документу и, получив его данные, либо обновляет переменные-члены класса «вид» или соответствующие элементы управления, либо объявляет часть представления недействительной, что заставляет функцию *OnDraw* перерисовать часть окна по данным документа. *OnUpdate* может выглядеть так:

```
void CMyView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
   CMyDocument* pMyDoc = GetDocument();
   CString lastName = pMyDoc->GetLastName();
   m_pNameStatic->SetWindowText(lastName);    // m_pNameStatic - переменная-член
                                              // CMyView
}
```

Вспомогательная информация передается через вызов *UpdateAllViews*. Исходная реализация *OnUpdate* объявляет недействительным все окно представления. В переопределенной ее версии вы можете объявить недействительным прямоугольник меньшего размера — в соответствии с информацией, полученной от *UpdateAllViews* (через последние два параметра).

Если функция *UpdateAllViews* класса *CDocument* вызывается с параметром *pSender*, указывающим на конкретный объект «вид», *OnUpdate* вызывается для всех представлений документа, *кроме* указанного в параметре.

Функция *CView::OnInitialUpdate*

Эта виртуальная функция класса *CView* вызывается при запуске приложения, а также при выборе команд New или Open меню File. Версия *OnInitialUpdate* в базовом классе *CView* просто вызывает *OnUpdate*. Переопределяя ее в своем производном

классе «вид», позаботьтесь, чтобы она вызывала *OnInitialUpdate* базового или *OnUpdate* производного класса.

Функция *OnInitialUpdate* производного класса пригодна для инициализации объекта «вид». При запуске программы каркас приложений вызывает *OnInitialUpdate* сразу после вызова *OnCreate* (если вы задействовали *OnCreate* в своем производном классе «вид»). *OnCreate* вызывается только раз, но *OnInitialUpdate* можно вызывать неоднократно.

Функция *CDocument::OnNewDocument*

Эту виртуальную функцию каркас приложений вызывает после создания объекта «документ» или выбора в меню File SDI-приложения команды New. Именно здесь удобнее всего инициализировать переменные-члены вашего класса «документ». Для производного класса документа MFC Application Wizard генерирует переопределенную функцию *OnNewDocument*. Обязательно оставьте в ней вызов функции базового класса.

Простейшее приложение в архитектуре «документ-вид»

Допустим, вам не нужно несколько представлений документа, но необходима поддержка каркаса приложений для работы с файлами. Тогда забудьте о функциях *UpdateAllViews* и *OnUpdate* и придерживайтесь следующей простой схемы.

1. В заголовочном файле производного класса документа (сгенерированном MFC Application Wizard) объявите переменные-члены, в которых хранятся основные данные программы. Объявите их открытыми или сделайте производный класс «вид» дружественным классу документа.
2. В производном классе «вид» переопределите виртуальную функцию *OnInitialUpdate*. Каркас приложений вызывает ее после инициализации или считывания с диска данных документа. (О работе с дисковыми файлами см. главу 16.) *OnInitialUpdate* должна обновить представление в соответствии с текущим содержанием документа.
3. Сделайте так, чтобы обработчики оконных и командных сообщений и функция *OnDraw* в производном классе «вид» имели прямой доступ к переменным-членам класса «документ», используя для этого функцию *GetDocument*.

   В нашей упрощенной среде «документ-вид» события разворачиваются в таком порядке:

Запуск программы	Конструируется объект *CMyDocument*.
	Конструируется объект *CMyView*.
	Создается окно представления.
	Вызывается *CMyView::OnCreate* (если она сопоставлена этому классу).
	Вызывается *CMyDocument::OnNewDocument*.
	Вызывается *CView::OnInitialUpdate*.
	Инициализируется объект «вид».
	Окно представления объявляется недействительным.
	Вызывается *CMyView::OnDraw*.

Пользователь редактирует данные	Функции класса *CMyView* обновляют переменные-члены *CMyDocument*.
Завершение работы программы	Уничтожается объект *CMyView*.
	Уничтожается объект *CMyDocument*.

Класс *CFormView*

Этот весьма полезный класс обладает многими свойствами немодального диалогового окна. Класс, производный от *CFormView*, как и от *CDialog*, связан с диалоговым ресурсом, определяющим параметры окна и список элементов управления. Класс *CFormView* поддерживает те же DDX- и DDV-функции обмена и проверки данных, что и в использовавших *CDialog* программах из главы 7.

Объект *CFormView* получает уведомляющие сообщения прямо от своих элементов управления, а также принимает командные сообщения от каркаса приложений. Очевидное отличие *CFormView* от *CDialog* — способность первого обрабатывать команды каркаса приложений, что упрощает управление окном представления из основного меню окна-рамки или через панель инструментов.

Внимание! Если диалоговое окно для *CFormView* сгенерировано MFC Application Wizard, его свойства задаются корректно, но, создавая его в редакторе диалоговых окон, *обязательно* задайте в диалоговом окне Dialog Properties свойства:

- Style = Child (дочернее окно);
- Border = None (без обрамления);
- Visible = флажок сброшен (изначально невидимо).

Класс *CFormView* — производный от *CView* (точнее, от *CScrollView*), а не от *CDialog*. Так что не надейтесь на присутствие функций-членов *CDialog*. В нем *нет* виртуальных функций *OnInitDialog*, *OnOK* и *OnCancel*. Функции-члены класса *CFormView* не вызывают *UpdateData* и DDX-функции. Вы сами должны заботиться о вызовах *UpdateData* (обычно в ответ на уведомления от элементов управления или на командные сообщения).

Хотя *CFormView* происходит не от *CDialog*, он построен на основе диалогового окна Windows. Поэтому вы можете использовать многие функции-члены класса *CDialog*, например, *GotoDlgCtrl* и *NextDlgCtrl*: надо лишь привести тип указателя на *CFormView* к указателю на *CDialog*. Показанный ниже оператор, извлеченный из функции-члена некоего класса, производного от *CFormView*, устанавливает фокус на заданный элемент управления. *GetDlgItem* — это функция класса *CWnd*, поэтому класс, производный от *CFormView*, ее наследует.

```
((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
```
[1]

MFC Application Wizard позволяет использовать *CFormView* в качестве базового для вашего класса «вид». В этом случае MFC Application Wizard сгенерирует пустое диалоговое окно с корректным набором стилей. Далее в окне Properties утилиты

---

[1] Весьма опасное приведение типа, которое работает только потому, что *GotoDlgCtrl* использует из класса *CDialog* лишь переменную-член *m_hWnd*. Эта переменная унаследована из *CWnd*, поэтому она есть и в классе *CFormView* . — *Прим. перев.*

Class View создайте обработчики уведомляющих сообщений от элементов управления, обработчики командных сообщений и сообщений обновления пользовательского интерфейса. (Подробнее об этом см. пример.) Кроме того, вы можете определить переменные-члены и критерии проверки.

Класс *CObject*

На вершине иерархии MFC-классов находится класс *CObject*. Большинство остальных классов наследует *корневому* классу *CObject*. Класс, производный от *CObject*, наследует ряд важных характеристик. Многие преимущества этого станут очевидны при чтении следующих глав.

В этой главе мы рассмотрим, как наследование от *CObject* позволяет задействовать объекты в организации вывода диагностической информации, а также включать их в *классы наборов* (collection class).

Диагностика

В MFC-библиотеке есть ряд полезных средств дампа диагностической информации. Эти средства активизируются, если выбрать конфигурацию сборки Debug, в случае же выбора конфигурации Release отображение диагностической информации отключается, и диагностический код не компонуется с программой. Весь диагностический вывод направляется в окно Output отладчика.

Совет Для очистки окна диагностического вывода поместите в него курсор, щелкните правой кнопкой и в контекстном меню выберите команду Clear All.

Макрос *TRACE*

Вы уже встречали данный макрос в предыдущих примерах. Операторы *TRACE* активизируются, если определена константа *_DEBUG* — это происходит в конфигурации Debug и когда переменной *afxTraceEnabled* присвоено значение *TRUE*. Оператор *TRACE* работает аналогично функции *printf* языка С, но полностью отключается в *финальной* (Release) версии программы. Вот типичный пример:

```
int nCount = 9;
CString strDesc("total");
TRACE("Count = %d, Description = %s\n", nCount, strDesc);
```

Хотя макрос *TRACE* и не рекомендуется (в документации предлагается использовать макрос *ATLTRACE*), он все еще доступен и прекрасно работает.

Объект *afxDump*

Эта альтернатива *TRACE* более годится для языка C++. MFC-объект *afxDump* принимает переменные из программы с использованием синтаксиса, аналогичного синтаксису объекта C++ потокового вывода *cout*. Применять сложные строки форматирования не требуется — формат вывода управляется переопределяемыми операторами. Вывод *afxDump* направляется туда же, куда и вывод *TRACE*, но объект

afxDump определен только в отладочной версии MFC-библиотеки. Вот типичный ориентированный на потоки диагностический оператор, который дает тот же результат, что и приведенный выше макрос *TRACE*:

```
int nCount = 9;
CString strDesc("total");
#ifdef _DEBUG
   afxDump << "Count = " << nCount
           << " , Description = " << strDesc << "\n";
#endif
```

Хотя и в *afxDump*, и в *cout* применяется одинаковый оператор вставки (<<), код их реализации различен. Объект *cout* — часть библиотеки iostream Visual C++, а *afxDump* — часть MFC-библиотеки. Не думайте, что какие-то возможности форматирования, обеспечиваемые *cout*, доступны при работе с *afxDump*.

У классов, не производных от *CObject*, таких как *CString*, *CTime* и *CRect*, есть свои переопределенные операторы вставки для объектов *CDumpContext*. Класс *CDumpContext*, экземпляром которого является *afxDump*, содержит переопределенные операторы вставки для базовых типов С++ (*int*, *double*, *char** и т. д.). Кроме того, в нем есть переопределенные операторы для ссылок и указателей на *CObject* — они-то и представляют для нас интерес.

Классы *CDumpContext* и *CObject*

Если оператор вставки класса *CDumpContext* принимает указатели и ссылки на *CObject*, он должен также принимать указатели и ссылки на производные классы. Рассмотрим тривиальный класс *CAction*, производный от *CObject*:

```
class CAction : public CObject
{
public:
   int m_nTime;
};
```

Что же происходит при выполнении следующего оператора?

```
#ifdef _DEBUG
   afxDump << action; // action - объект класса CAction
#endif
```

А вот что. Вызывается виртуальная функция *CObject::Dump*. Если вы не переопределили ее для *CAction*, то многого от нее не получите — разве что адрес объекта. Переопределив же *Dump*, можно получить сведения о внутреннем состоянии интересующего нас объекта. Взглянем на функцию *CAction::Dump*:

```
#ifdef _DEBUG
void CAction::Dump(CDumpContext& dc) const
{
   CObject::Dump(dc);  // всегда вызывайте функцию базового класса
   dc << "\ntime = " << m_nTime << "\n";
}
#endif
```

Функция *Dump* базового класса (*CObject*) выводит примерно такую строку:

```
a CObject at $4115D4
```

Если в определении класса *CAction* использован макрос *DECLARE_DYNAMIC*, а в реализации — *IMPLEMENT_DYNAMIC*, то при выводе диагностической информации вы получите имя данного класса:

```
a CAction at $4115D4
```

даже если оператор отладочного вывода выглядит так:

```
#ifdef _DEBUG
   afxDump << (CObject&) action;
#endif
```

Вместе эти два макроса вставляют в ваш класс, производный от *CObject*, специальный код библиотеки MFC периода выполнения. При наличии такого кода программа в период выполнения может определить имя класса объекта (например, для отладочного дампа) и получить сведения об иерархии классов.

Примечание Пары макросов (*DECLARE_SERIAL, IMPLEMENT_SERIAL*) и (*DECLARE_DYNCREATE, IMPLEMENT_DYNCREATE*) обеспечивают ту же поддержку периода выполнения, что и пара макросов (*DECLARE_DYNAMIC, IMPLEMENT_DYNAMIC*).

Автоматическая диагностика неуничтоженных объектов

Если программа собирается в конфигурации Debug, после завершения программы каркас приложений сообщает обо всех неуничтоженных объектах. Эта диагностика очень полезна; но все же аккуратно удаляйте *все* объекты — даже те, что обычно сами исчезают при завершении программы. Такая очистка — показатель хорошего стиля программирования.

Код, добавляющий отладочную информацию в выделенные блоки памяти, теперь находится не в MFC-библиотеке, а в отладочной версии С-библиотеки периода выполнения (CRT). Если вы решите загружать MFC динамически, то вместе с DLL-модулями MFC-библиотеки будет загружаться и MSVCRTD.DLL. Если вы добавите в начало CPP-файла строку:

```
#define new DEBUG_NEW
```

CRT-библиотека покажет имя файла и номер строки, в которой был распределен данный блок памяти. Эту строку MFC Application Wizard вставляет в начало всех сгенерированных им CPP-файлов.

Создание оконных подклассов для расширенного управления вводом данных

А если требуется поле ввода (в диалоговом окне или окне представления в виде формы), допускающее ввод только чисел? Нет ничего проще: установите стиль Number в окне свойств элемента управления. Однако если нуж-

но исключить цифровые символы или изменить регистр букв, придется немного попрограммировать.

Библиотека MFC позволяет легко изменить поведение любого стандартного элемента управления, в том числе поля ввода. На это есть два способа. Можно создать свои классы, производные от *CEdit*, *CListBox* и т. д. (с собственными функциями-обработчиками сообщений), а затем динамически формировать объекты элементов управления. Или, как это сделал бы программист в Win32, зарегистрировать специальный оконный класс и интегрировать его в файл ресурсов проекта, используя текстовый редактор. Однако ни один из указанных способов не позволяет размещать такие элементы управления в диалоговом ресурсе через редактор диалоговых окон.

Проще всего изменить поведение элемента управления инструментом MFC-библиотеки, предназначенным для создания *оконных подклассов* (window subclassing). При использовании редактора диалоговых окон в диалоговом ресурсе размещается обычный элемент управления, а потом на C++ пишется новый класс, содержащий обработчики сообщений для событий, которые вы хотите обрабатывать сами. Вот как создать подкласс поля ввода:

1. Используя редактор диалоговых окон, расположите поле ввода в диалоговом ресурсе. Пусть его идентификатор — *IDC_EDIT1*.
2. Создайте новый класс (например, *CNonNumericEdit*), производный от *CEdit*. Напишите обработчик сообщения *WM_CHAR*, скажем, такой:

```
[illegible]
```

3. В заголовке производного класса диалогового окна или формы объявите переменную-член класса *CNonNumericEdit*:

```
[illegible]
```

4. Если вы работаете с классом диалогового окна, добавьте в переопределенную функцию *OnInitDialog* строку:

```
[illegible]
```

5. Если же вы работаете с классом формы, добавьте в переопределенную функцию *OnInitialUpdate*:

```
[illegible]
```

Функция-член *SubclassDlgItem* класса *CWnd* гарантирует, что все сообщения, прежде чем дойти до встроенной оконной процедуры элемента управления, пройдут через организуемую каркасом приложений систему маршрутизации сообщений. Такой прием называется *динамическим созданием*

см. след. стр.

подклассов (dynamic subclassing) и подробно описан в Technical Note #14 документации по библиотеке MFC.

Приведенный код только проверяет символы и отвергает недопустимые. Если же надо изменить значение символа, обработчик должен вызвать *CWnd::DefWindowProc*, что позволит обойти стандартное поведение MFC, предусматривающее хранение значения параметров в переменных объекта-потока. Вот пример обработчика, который переводит все буквы в верхний регистр:

```
[illegible]
{
    [illegible]
        [illegible]
    }
    [illegible]
        [illegible]
}
```

Оконные подклассы можно использовать и для обработки *возвращенных* (reflected) сообщений. Если оконный MFC-класс не сможет сопоставить сообщению от одного из дочерних элементов управления обработчик, каркас приложений возвратит сообщение обратно этому элементу. Подробнее об этом см. Technical Note #62 документации по библиотеке MFC.

Если вам требуется поле ввода, скажем, с желтым фоном, создайте класс *CYellowEdit*, производный от *CEdit*, и в окне Properties вида Class View сопоставьте в *CYellowEdit* сообщению =WM_CTLCOLOR обработчик (знак «равно» в Class View перед именем сообщения говорит о том, что речь идет о возвращенном сообщении). Обработчик практически не отличается от обработчика обычного сообщения WM_CTLCOLOR (см. главу 7). (Переменная *m_hYellowBrush* инициализируется в конструкторе класса элемента управления.)

```
[illegible]
{
    [illegible]
    [illegible]
}
```

Пример Ex15a: простое взаимодействие между документом и представлением

Первый из двух примеров этой главы (рис. 15-1) иллюстрирует простейший случай взаимодействия «документ-вид». У класса документа *CEx15aDoc*, производного от *CDocument*, есть единственный внедренный объект *CStudent*, который представляет одну запись о студенте, состоящую из имени (*CString*) и целочисленного балла. Класс «вид» *CEx15aView*, производный от *CFormView*, отображает на экране запись о студенте и содержит поля ввода имени и балла. Кнопка по умолчанию Enter обновляет данные документа согласно содержимому полей ввода.

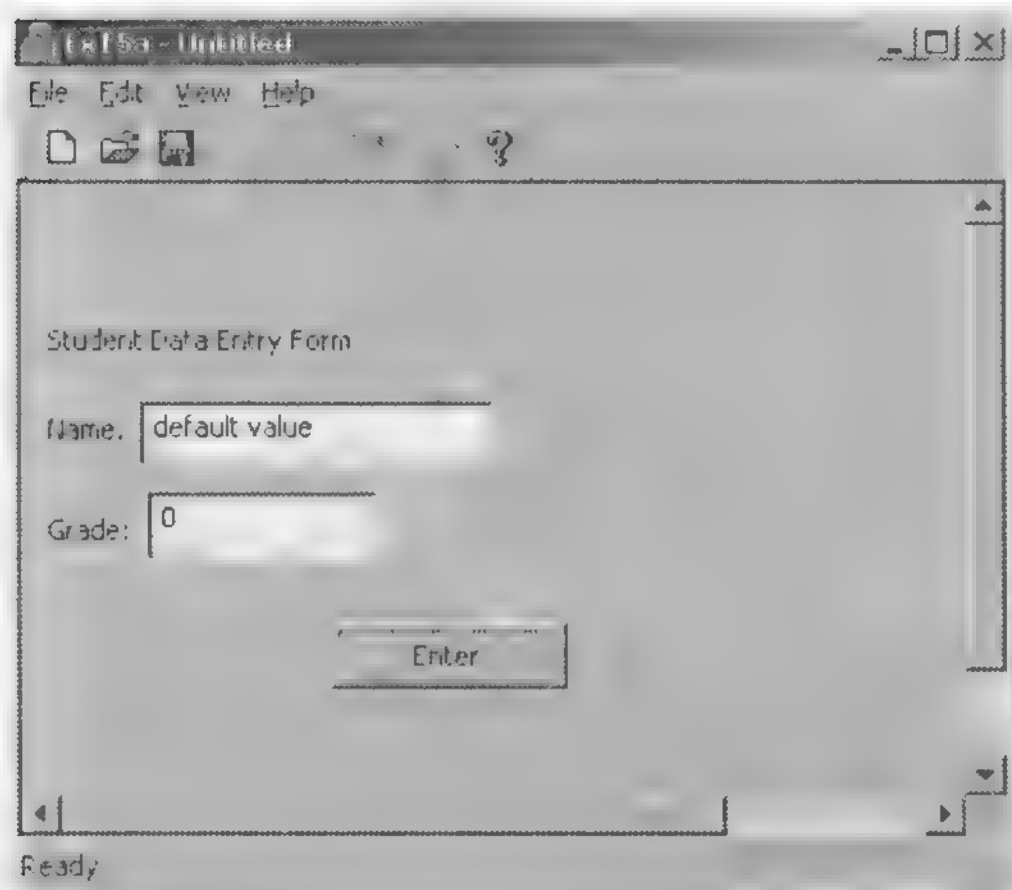


Рис. 15-1. *Программа Ex15a в действии*

Код класса *CStudent* приведен в листинге. Большинство возможностей данного класса предназначено для программы Ex15a, но некоторые из них пригодятся как в Ex15b, так и в программах-примерах главы 16. А пока обратите внимание на переменные-члены, конструктор по умолчанию и на объявление функции *Dump*. Макросы *DECLARE_DYNAMIC* и *IMPLEMENT_DYNAMIC* обеспечивают вывод для класса.

Student.h

```
// student.h

#ifndef _INSIDE_VISUAL_CPP_STUDENT
#define _INSIDE_VISUAL_CPP_STUDENT
class CStudent : public CObject
{
   DECLARE_DYNAMIC(CStudent)
public:
   CString m_strName;
   int m_nGrade;

   CStudent()
   {
      m_nGrade = 0;
   }

   CStudent(const char* szName, int nGrade) : m_strName(szName)
   {
      m_nGrade = nGrade;
   }

   CStudent(const CStudent& s) : m_strName(s.m_strName)
   {
      // конструктор копии
```

см. след. стр.

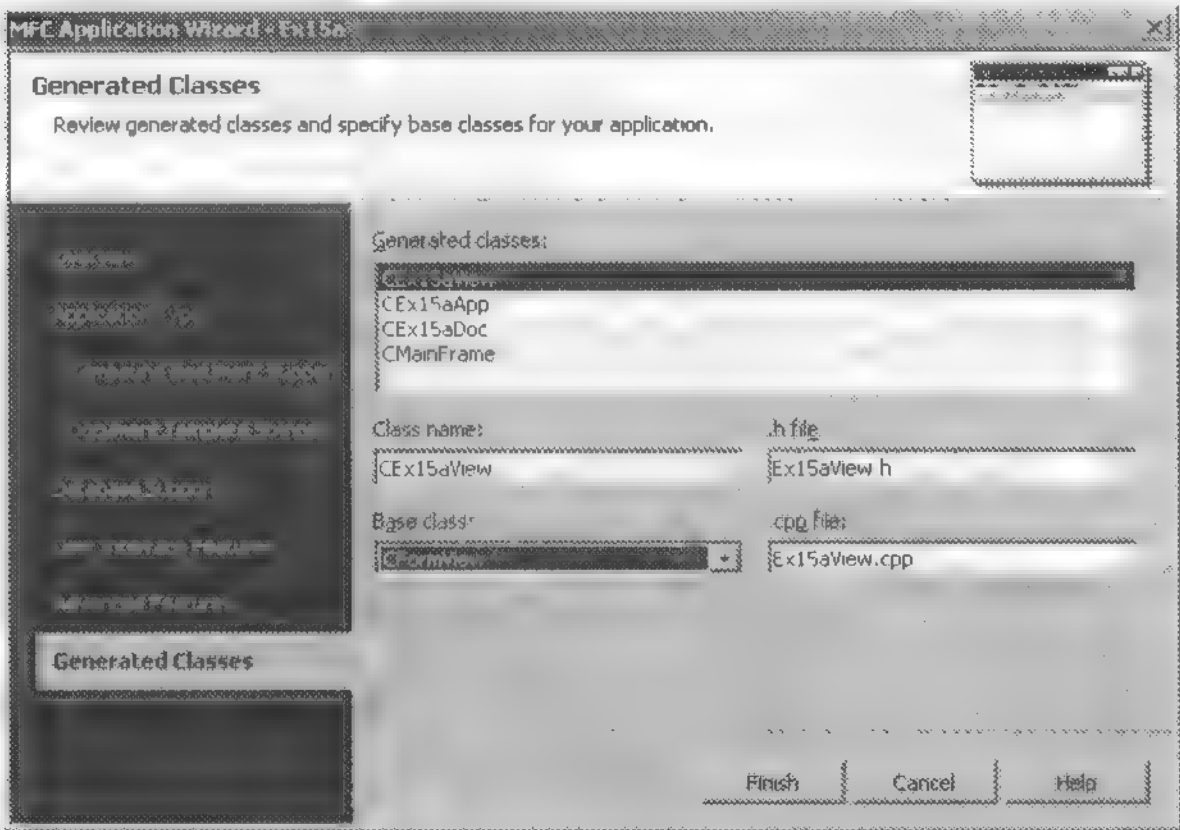
```
[illegible]
```

Student.cpp

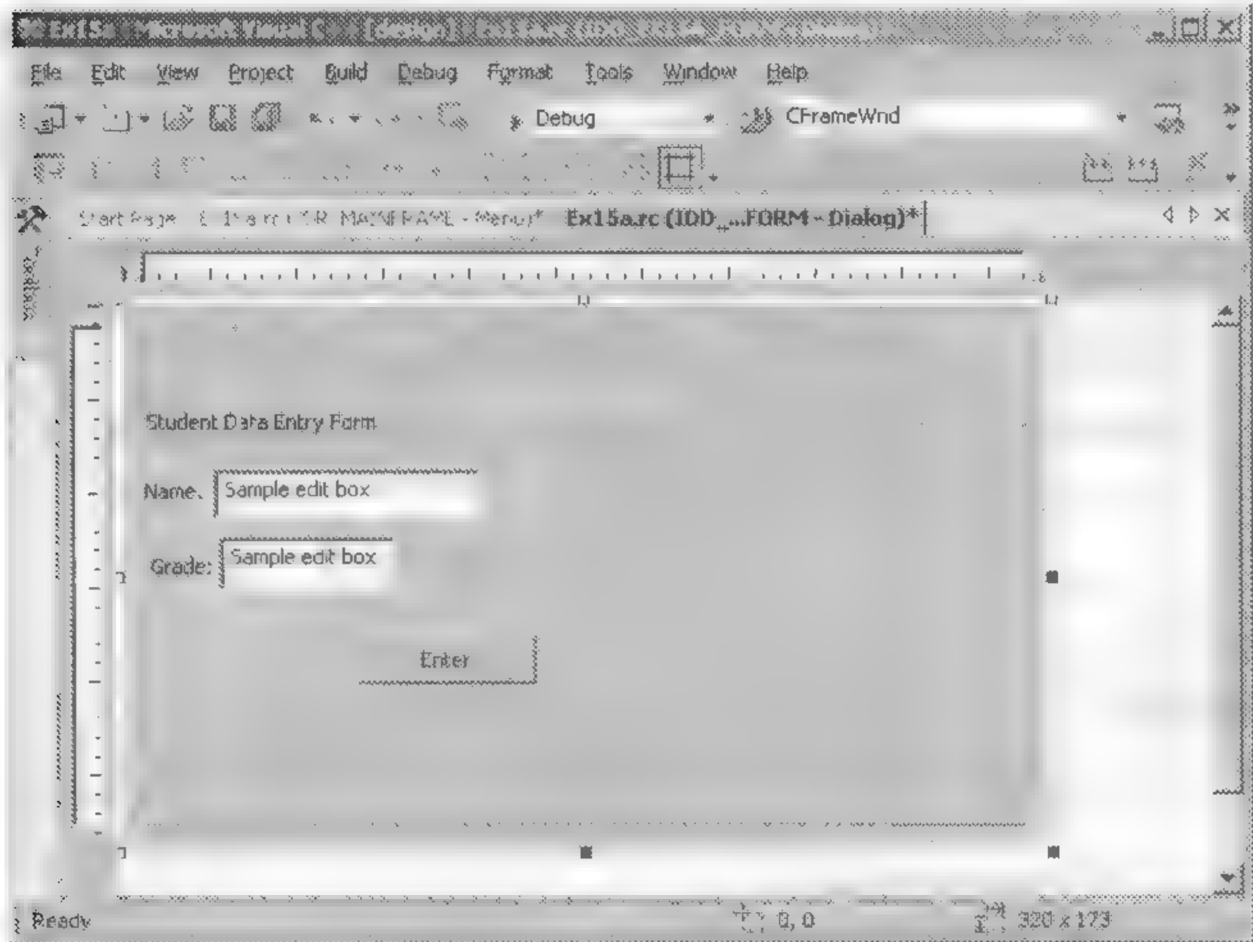
```
[illegible]
```

Итак, создаем программу Ex15a.

1. **Используя MFC Application Wizard, создайте проект Ex15a.** На странице Application Type мастера установите переключатель в положение Single document, а на странице Generated Classes в качестве базового выберите класс *CFormView*:



2. **Используя редактор меню, замените команды в меню Edit.** Удалите прежнее содержимое меню Edit и вставьте команду Clear All. Используйте константу по умолчанию *ID_EDIT_CLEARALL*, назначенную каркасом приложений.
3. **В редакторе диалоговых окон измените диалоговое окно *IDD_EX15A_FORM*.** Открыв диалоговый ресурс *IDD_EX15A_FORM*, сгенерированный мастером MFC Application Wizard, добавьте элементы управления:



Убедитесь, что свойства в окне Properties заданы так: Style = Child, Border = None и Visible = False. Присвойте элементам управления идентификаторы:

Элемент управления	Идентификатор
Поле ввода Name	*IDC_NAME*
Поле ввода Grade	*IDC_GRADE*
Кнопка Enter	*IDC_ENTER*

4. **В окне Properties утилиты Class View добавьте обработчики сообщений класса *CEx15aView*.** Выберите класс *CEx15aView* в окне Class View и добавьте следующие обработчики сообщений. Оставьте имена функций по умолчанию.

Идентификатор объекта	Сообщение	Имя функции-члена
IDC_ENTER	*BN_CLICKED*	*OnBnClickedEnter*
ID_EDIT_CLEARALL	*COMMAND*	*OnEditClearall*
ID_EDIT_CLEARALL	*UPDATE_COMMAND_UI*	*OnUpdateEditClearall*

5. **В окне Properties утилиты Class View добавьте переменные для *CEx15aView*.** В окне Class View щелкните правой кнопкой класс *CEx15aView* и в контекстном меню выберите Add Variable. Добавьте переменные:

Элемент управления	Имя переменной	Категория	Тип переменной
IDC_GRADE	*m_nGrade*	*Value*	*int*
IDC_NAME	*m_strName*	*Value*	*CString*

Задайте для *m_nGrade* минимальное значение 0, максимальное — 100. Заметьте, что Add Member Variable Wizard генерирует код проверки данных, вводимых пользователем.

6. **Добавьте прототип вспомогательной функции *UpdateControlsFromDoc*.** В окне Class View щелкните правой кнопкой класс *CEx15aView* и в контекстном меню выберите команду Add Function. Заполните диалоговую форму, чтобы добавить функцию:

```
private:
    void UpdateControlsFromDoc();
```

7. **Отредактируйте файл Ex15aView.cpp.** MFC Application Wizard создал заготовку функции *OnInitialUpdate*, а Add Member Function Wizard — функции *UpdateControlsFromDoc*. Последняя представляет собой закрытую *вспомогательную* (helper) функцию-член, которая переносит данные из документа в переменные-члены *CEx15aView*, а затем в элементы управления диалогового окна. Отредактируйте код так:

```
void CEx15aView::OnInitialUpdate()
{   // вызывается при запуске
    CFormView::OnInitialUpdate();
    UpdateControlsFromDoc();
}
void CEx15aView::UpdateControlsFromDoc(void)
{   // вызывается из OnInitialUpdate и OnEditClearall
    CEx15aDoc* pDoc = GetDocument();
    m_nGrade = pDoc->m_student.m_nGrade;
    m_strName = pDoc->m_student.m_strName;
    UpdateData(FALSE); // вызов DDX
}
```

OnBnClickedEnter заменяет функцию *OnOK*, которую можно было бы ожидать в классе диалогового окна. Она передает данные из полей ввода в переменные-члены объекта «вид», а потом в документ. Добавьте выделенный код:

```
void CEx15aView::OnBnClickedEnter()
{
    CEx15aDoc* pDoc = GetDocument();
    UpdateData(TRUE);
    pDoc->m_student.m_nGrade = m_nGrade;
    pDoc->m_student.m_strName = m_strName;
}
```

В сложной программе с несколькими представлениями данных команда Clear All передавалась бы прямо документу. В нашем же простом примере она доставляется объекту «вид». Обработчик сообщения обновления пользовательского интерфейса отключает элемент меню, если объект *CStudent* в документе уже пуст. Введите выделенный код:

```
void CEx15aView::OnEditClearall()
{
    GetDocument()->m_student = CStudent(); // "пустой" объект
    UpdateControlsFromDoc();
}
void CEx15aView::OnUpdateEditClearall(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(GetDocument()->m_student != CStudent());
// пустой?
}
```

8. **Добавьте в проект Ex15a файлы класса *CStudent*.** Скопируйте файлы Student.h и Student.cpp с компакт-диска. В Visual C++ .NET из меню Project выберите команду Add Existing Item и в списке файлов — Student.h и Student.cpp. Щелкните OK. Visual C++ .NET добавит имена этих файлов в проект, так что при сборке проекта они будут автоматически скомпилированы.
9. **Добавьте переменную типа *CStudent* в класс *CEx15aDoc*.** Это можно сделать, используя ClassView, тогда директива *#include* будет добавлена автоматически.

```
public:
    CStudent m_student;
```

Конструктор класса *CStudent* вызывается при создании объекта «документ», а деструктор вызывается автоматически при уничтожении документа.

10. **Отредактируйте файл Ex15aDoc.cpp.** Для инициализации объекта *CStudent* используйте конструктор *CEx15aDoc*:

```
CEx15aDoc::CEx15aDoc() : m_student("default value", 0)
{
    TRACE("Document object constructed\n");
}
```

Мы не определим, правильно ли работает Ex15a, пока не отобразим содержимое документа по ее завершении. Для этого применим деструктор, который вызовет функцию *Dump* документа, а та — функцию *CStudent::Dump*.

```
CEx15aDoc::~CEx15aDoc()
{
#ifdef _DEBUG
   Dump(afxDump);
#endif // _DEBUG
}

void CEx15aDoc::Dump(CDumpContext& dc) const
{
   CDocument::Dump(dc);
   dc << "\n" << m_student << "\n";
}
```

11. **Соберите и протестируйте приложение Ex15a.** Введите какое-нибудь имя и балл, а затем щелкните Enter. Закройте приложение. Появились ли в окне Debug сообщения, аналогичные показанным ниже?

```
a CEx15aDoc at $411580
m_strTitle = Untitled
m_strPathName =
m_bModified = 0
m_pDocTemplate = $4113A0

a CStudent at $4115D4
m_strName = Sullivan, Walter
m_nGrade = 78
```

Примечание Чтобы увидеть эти сообщения, скомпилируйте программу в отладочной конфигурации (Debug) и запустите ее под управлением отладчика.

Усложненное взаимодействие документа и представления

В программе, поддерживающей множественное представление данных, взаимодействие «документ-вид» сложнее, чем в Ex15a. Основная проблема в том, что пользователь редактирует данные в одном окне представления, а остальные окна представления надо обновлять, чтобы отразить внесенные изменения. Здесь понадобятся функции *UpdateAllViews* и *OnUpdate*, поскольку документ будет выступать в качестве координирующего центра при обновлении окон представления. При разработке придерживайтесь следующей схемы.

1. В созданном MFC Application Wizard заголовочном файле производного класса документа объявите переменные-члены документа. Их можно сделать закрытыми, а затем определить функции-члены для доступа к ним или объявить класс «вид» дружественным классу «документ».
2. В производном классе «вид» через окно Properties утилиты Class View, переопределите виртуальную функцию-член *OnUpdate*. Каркас приложений вызывает ее при любом изменении данных в документе. *OnUpdate* должна изменять объект «вид» в соответствии с текущим содержанием документа.

3. Проанализируйте командные сообщения. Для каждого определите, к чему оно относится: к документу или к его представлению. (Пример команды, относящейся к документу, — Clear All в меню Edit.) Теперь сопоставьте команды соответствующим классам.
4. Предусмотрите обновление данных в документе соответствующими функциями-обработчиками командных сообщений в производном классе «вид». Позаботьтесь, чтобы все они перед завершением обращались к *CDocument::UpdateAllViews.* Для доступа к документу применяйте версию функции-члена *CView::GetDocument*, обеспечивающую безопасное приведение типов.
5. Запрограммируйте обновление данных документа соответствующими функциями-обработчиками командных сообщений в производном классе документа. Все они перед завершением тоже должны вызывать *CDocument::UpdateAllViews.*

Последовательность событий для сложного взаимодействия «документ-вид» такова.

Запускается приложение	Создается объект *CMyDocument.*
	Создается объект *CMyView.*
	Создаются другие объекты «вид».
	Создаются окна представлений.
	Вызывается *CMyView::OnCreate* (если она сопоставлена).
	Вызывается *CDocument::OnNewDocument.*
	Вызывается *CView::OnInitialUpdate.*
	Вызывается *CMyView::OnUpdate.*
	Инициализируется объект «вид».
Пользователь выбирает команду, относящуюся к представлению данных	Функции класса *CMyView* обновляют переменные-члены класса *CMyDocument.*
	Вызывается функция *CDocument::UpdateAllViews.*
	Вызывается функция *OnUpdate* для других объектов «вид».
Пользователь выбирает команду, относящуюся к документу	Функции *CMyDocument* обновляют переменные-члены.
	Вызывается функция *CDocument::UpdateAllViews.*
	Вызывается функция CMyView::OnUpdate.
	Вызываются функции *OnUpdate* для других объектов «вид».
Пользователь завершает приложение	Уничтожаются объекты «вид».
	Уничтожается объект *CMyDocument.*

Функция *CDocument::DeleteContents*

Иногда нужна функция, способная удалить содержимое документа. Вы могли бы написать свою закрытую функцию-член, но каркас приложений уже определил для этого виртуальную функцию *DeleteContents* в классе *CDocument.* Каркас приложений вызывает вашу переопределенную функцию *DeleteContents* при закрытии документа и, как вы увидите в главе 16, в ряде других случаев.

Класс набора *CObList*

Познакомившись с классами наборов, вы наверняка удивитесь, как до сих пор без них обходились. Один из представителей этого семейства — *CObList*. Освоив его, вы легко разберетесь в *классах списков* (list class), *массивов* (array class) и *словарей* (map class).

Может показаться, что наборы — это что-то новое, но в языке С всегда поддерживалась одна из их разновидностей — массивы. Размер массивов в языке С фиксирован, они не поддерживают вставку новых элементов. Программисты на С писали библиотеки функций для других наборов, включая *связанные списки* (linked lists), *динамические массивы* (dynamic array) и *индексируемые словари* (indexed dictionary). В языке С++ есть очевидная и более удачная (в сравнении с библиотекой С-функций) альтернатива по реализации наборов — это классы. Например, объект-список корректно инкапсулирует внутренние структуры данных списка.

Класс *CObList* поддерживает упорядоченные списки указателей на объекты классов, производных от *CObject*. Другой MFC-класс наборов, *CPtrList*, вместо указателей на *CObject* хранит указатели *void*. Почему бы не задействовать его вместо *CObList*? Класс *CObList* имеет ряд преимуществ и при выводе диагностической информации, и при *сериализации* (serialization) (см. главу 16). Одно из важных свойств *CObList* — способность хранить *смешанные указатели* (mixed pointers). Иначе говоря, набор *CObList* может одновременно содержать указатели как на объекты *CStudent*, так и на объекты *CTeacher*, при условии что оба класса наследуют *CObject*.

Применение класса *CObList* для создания списков типа FIFO

Один из простейших примеров применения объекта *CObList* — добавление новых элементов в конец и удаление элементов из начала списка. Элемент, первым внесенный в список, первым и извлекается из него [принцип FIFO (First In, First Out) — «первым вошел, первым вышел»]. Допустим, элементы списка — объекты созданного вами класса *CAction*, производного от *CObject*. Следующая программа работает в режиме командной строки и заносит в список 5 элементов, а затем извлекает их в том же порядке.

```
#include <afx.h>
#include <afxcoll.h>

class CAction : public CObject
{
private:
   int m_nTime;
public:
   CAction(int nTime) { m_nTime = nTime; }    // конструктор запоминает
                                              // целочисленные показания времени
   void PrintTime() { TRACE("time = %d\n", m_nTime); }
};
int main()
{
```

```
   CAction* pAction;
   CObList actionList; // список CAction создается на стеке
   int i;
   // вставляем объекты CAction в порядке от 0 до 4
   for(i = 0; i < 5; i++) {
      pAction = new CAction(i);
      actionList.AddTail(pAction); // приведение типа для pAction не требуется
   }

   // извлекаем и удаляем объекты CAction в порядке от 0 до 4
   while (!actionList.IsEmpty()) {
      pAction =                                  // для возвращаемого значения
         (CAction*) actionList.RemoveHead();     // необходимо приведение типа
      pAction->PrintTime();
      delete pAction;
   }

   return 0;
}
```

В программе сначала создается объект *actionList* класса *CObList*. Затем функция-член *CObList::AddTail* добавляет в конец списка указатели на вновь создаваемые объекты *pAction*. Преобразовывать тип для *pAction* не нужно, так как параметром *AddTail* служит указатель на *CObject*, а *pAction* — указатель на производный от него класс.

Далее указатели на объекты *CAction* извлекаются из начала списка, и объекты удаляются. В данном случае для возвращаемого значения *RemoveHead* требуется приведение типа, поскольку эта функция возвращает указатель на класс *CObject*, расположенный в иерархии классов *выше* класса *CAction*.

Когда вы извлекаете указатель на объект из набора, сам объект автоматически не уничтожается. Удалять объекты *CAction* нужно оператором *delete*.

Перебор элементов *CObList*: переменная типа *POSITION*

Допустим, вам надо «пройти» все элементы списка. Класс *CObList* предусматривает функцию-член *GetNext*, которая возвращает указатель на «следующий» элемент списка, но способ ее применения несколько необычен. *GetNext* принимает целочисленный параметр типа *POSITION*, который представляет собой 32-разрядную переменную. Она содержит внутреннее представление положения в списке извлекаемого элемента. Так как *POSITION* передается по ссылке (&), функция может изменить его значение.

GetNext:

1. возвращает указатель на «текущий» элемент списка, который определяется передаваемым ей значением *POSITION*;
2. изменяет значение параметра *POSITION* так, чтобы он соответствовал положению следующего элемента списка.

Так выглядит цикл с *GetNext* для списка из предыдущего примера.

```
CAction* pAction;
POSITION pos = actionList.GetHeadPosition();
while (pos != NULL) {
    pAction = (CAction*) actionList.GetNext(pos);
    pAction->PrintTime();
}
```

Теперь допустим, что у вас есть интерактивное Windows-приложение, в котором для перемещения вперед и назад по элементам списка служат кнопки панели инструментов. Для выбора очередного элемента *GetNext* не годится, так как она всегда «увеличивает» значение переменной *POSITION*, а предугадать, следующий или предыдущий элемент потребуется пользователю, просто невозможно. Взгляните на пример обработчика командного сообщения в классе «вид», который выбирает следующий элемент списка. Здесь *m_actionList* — объект типа *CObList*, внедренный в класс *CMyView*, а *m_position* — переменная-член типа *POSITION*, хранящая положение текущего элемента списка.

```
CMyView::OnCommandNext()
{
    POSITION pos;
    CAction* pAction;

    if ((pos = m_position) != NULL) {
        m_actionList.GetNext(pos);
        if (pos != NULL) { // pos равно NULL в конце списка
            pAction = (CAction*) actionList.GetAt(pos);
            pAction->PrintTime();
            m_position = pos;
        }
        else {
            AfxMessageBox("End of list reached");
        }
    }
}
```

Чтобы увеличить переменную положения, вызывают *GetNext*, а чтобы получить элемент — функцию *CObList::GetAt*. Переменная *m_position* обновляется, только если не достигнут конец списка.

Класс-шаблон набора *CTypedPtrList*

Класс *CObList* отлично работает, когда нужен набор, содержащий смешанные указатели. Однако если требуется набор с безопасным приведением типов, содержащий указатели только на один тип объектов, используйте классы-шаблоны наборов указателей из MFC-библиотеки. Пример такого класса — *CTypedPtrList*. *Шаблоны* (templates) — относительно новый элемент C++, появившийся в Microsoft Visual C++ 2.0. *CTypedPtrList* — это класс-шаблон, который применяется для создания списка указателей на объекты любого заданного класса. Не вдаваясь в подробности, скажем, что шаблон применяют для создания нового класса списка, производного от *CPtrList* или *CObList*.

Объект для указателей на *CAction* объявляется так:

```
CTypedPtrList< CObList, CAction* > m_actionList;
```

Первый параметр — это базовый класс набора, второй — тип параметров и возвращаемых значений. В качестве базовых допускаются только классы *CPtrList* и *CObList*, так как других классов наборов указателей в библиотеке MFC нет. Если вы храните объекты классов, производных от *CObject*, используйте как базовый класс *CObList*, в противном случае — *CPtrList*.

При описанном выше использовании шаблона компилятор гарантирует, что все функции-члены списка возвращают указатель *CAction*. Таким образом, можно писать:

```
pAction = m_actionList.GetAt(pos); // приведение типа не требуется
```

Чтобы слегка упростить запись, прибегнем *typedef* и сгенерируем подобие самостоятельного класса:

```
typedef CTypedPtrList<CObList, CAction*> CActionList;
```

и тогда *m_actionList* можно объявить так:

```
CActionList m_actionList;
```

Диагностика и классы наборов

Функция *Dump* для *CObList* и других классов наборов обладает весьма полезным свойством. Вызвав *Dump* для какого-либо объекта-набора, вы получите сведения обо всех объектах набора. Если в классах объектов, составляющих набор, применены макросы *DECLARE_DYNAMIC* и *IMPLEMENT_DYNAMIC*, в информации появится имя класса всех объектов.

По умолчанию функции *Dump* для наборов выводят только имена классов и адреса объектов в наборе. Чтобы функции *Dump* для наборов вызывали функцию *Dump* для каждого элемента набора, где-то в начале программы надо сделать вызов:

```
#ifdef _DEBUG
   afxDump.SetDepth(1);
#endif
```

Тогда оператор:

```
#ifdef _DEBUG
   afxDump << actionList;
#endif
```

будет выводить информацию примерно так:

```
a CObList at $411832
with 4 elements
   a CAction at $412CD6
time = 0
   a CAction at $412632
time = 1
   a CAction at $41268E
```

```
time = 2
   a CAction at $4126EA
time = 3
```

Если набор содержит смешанные указатели, для класса объекта вызывается виртуальная функция *Dump*, и выводится имя соответствующего класса.

Пример Ex15b: SDI-приложение с множественными представлениями

Этот пример расширяет программу Ex15a. Вот перечень основных отличий.

- Документ содержит не один, а список объектов *CStudent*. (Теперь вы понимаете, почему мы создали класс *CStudent* вместо того, чтобы сделать *m_strName* и *m_nGrade* переменными-членами класса «документ».)
- Кнопки на панели инструментов позволяют перемещаться по списку.
- Структура программы допускает создание дополнительных представлений данных. Команда Clear All теперь передается объекту «документ», и поэтому в игру вступают функции *UpdateAllViews* для документа и *OnUpdate* для его представления.
- Особый код для представления информации о студентах изолирован, чтобы класс *CEx15bView* можно было впоследствии трансформировать в базовый, содержащий только универсальный код. Производные классы могут переопределять отдельные функции для работы со списками объектов, характерными для конкретного приложения.

Окно программы Ex15b (рис. 15-2) отличается от окна программы Ex15a (рис. 15-1). Кнопки активны, только когда это допустимо. Например, в конце списка становится неактивной кнопка Next со стрелкой вниз.

На панели инструментов расположены кнопки:

Кнопка	Назначение
	Перейти на первую запись
	Перейти на последнюю запись
	Перейти на предыдущую запись
	Перейти на следующую запись
	Удалить текущую запись
	Вставить новую запись

Кнопка Clear в окне представления очищает поля ввода Name и Grade. Команда Clear All в меню Edit удаляет из списка все записи и очищает поля ввода в окне представления.

На этот раз мы не станем давать пошаговые инструкции. Поскольку объем кода увеличился, просто приведем листинг отдельных частей и требования к ресурсам. Дополнительные фрагменты кода и те места, где надо внести изменения в код, сгенерированный MFC Application Wizard и мастерами, доступными из окна Class

View, выделены. Операторы *TRACE* позволят наблюдать за ходом выполнения программы в окне отладки.

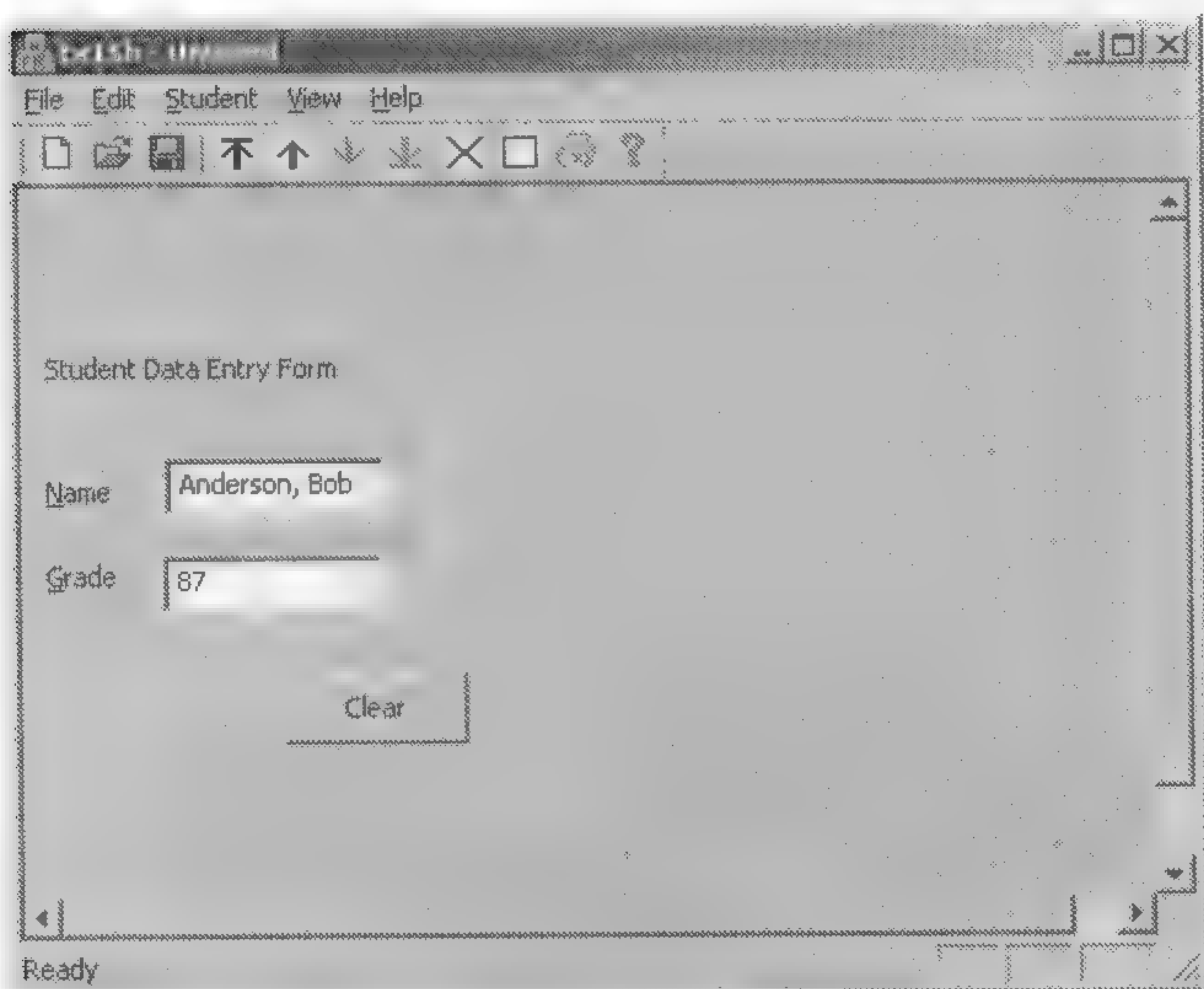


Рис. 15-2. *Программа Ex15b в действии*

Требования к ресурсам

В данном разделе описаны заданные в файле Ex15b.rc ресурсы.

Панель инструментов

Чтобы создать панель инструментов (рис. 15-2), нужно удалить кнопки Edit Cut, Copy и Paste (четвертую, пятую и шестую слева) и заменить их шестью новыми. Для создания некоторых кнопок применяется команда Flip Vertical из меню Image, а в файле Ex15b.rc определяется связь между идентификаторами команд и кнопками панели инструментов.

Меню Student

Присутствие в меню пунктов, соответствующих новым кнопкам панели инструментов, вообще-то необязательно. (Окно Properties средства Class View позволяет создавать обработчики команд панели инструментов так же просто, как и команды меню.) Но большинство Windows-приложений позволяет вызывать все команды через меню, поэтому лучше не обманывать ожиданий пользователей.

Меню Edit

В меню Edit операции с буфером обмена заменены пунктом Clear All (см. п. 2 примера Ex15a).

Диалоговый шаблон *IDD_EX15B_FORM*

Диалоговый шаблон *IDD_EX15B_FORM*, приведенный здесь, напоминает шаблон из примера Ex15a (рис. 15-1) за исключением того, что кнопку Enter заменила кнопка Clear.

Вот идентификаторы элементов управления:

Элемент управления	Идентификатор
Поле ввода Name	*IDC_NAME*
Поле ввода Grade	*IDC_GRADE*
Кнопка Clear	*IDC_CLEAR*

Стили элементов управления — такие же, как и у их аналогов из примера Ex15a.

Требования к коду

Файлы и классы в проекте Ex15b.

Заголовочный файл	Файл исходного кода	Классы	Описание
Ex15b.h	Ex15b.cpp	*CEx15bApp*	Класс приложения (создан MFC Application Wizard)
		CAboutDlg	Диалоговое окно About
MainFrm.h	MainFrm.cpp	*CMainFrame*	Основное окно-рамка SDI-приложения
Ex15bDoc.h	Ex15bDoc.cpp	*Ex15bDoc*	Документ приложения
Ex15b.h	Ex15b.cpp	*Ex15bView*	Класс «вид» — форма, представляющая информацию о студенте (производный от *CFormView*)
Student.h	Student.cpp	*CStudent*	Запись о студенте (как и в Ex15a)
StdAfx.h	StdAfx.cpp		Включает стандартные, предварительно откомпилированные заголовочные файлы

CEx15bApp

Файлы Ex15b.cpp и Ex15b.h — стандартные, сгенерированные MFC Application Wizard.

CMainFrame

Код этого класса в файле MainFrm.cpp — также стандартный результат работы MFC Application Wizard.

CStudent

Код тот же, что и в Ex15a, только в конец Student.h добавлена строка:

```
typedef CTypedPtrList< CObList, CStudent* > CStudentList;
```

Примечание Классы шаблонов наборов MFC требуют наличия в StdAfx.h оператора:

```
#include <afxtempl.h>
```

CEx15bDoc

Класс *CStudentDoc* сначала сгенерирован MFC Application Wizard. А вот код программы-примера Ex15b:

Ex15bDoc.h

```
[illegible]
[illegible]

[illegible]

#include "student.h"

[illegible]
[illegible]
[illegible]
    [illegible]
    [illegible]

[illegible]
[illegible]
    CStudentList* GetList() {
        return &m_studentList;
    }
[illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
    [illegible]

[illegible]
[illegible]
    [illegible]
[illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]

[illegible]
[illegible]
    [illegible]

private:
    CStudentList m_studentList;

[illegible]
```

Ex15bDoc.cpp

```
[illegible]
[illegible]

[illegible]
[illegible]

[illegible]

[illegible]
[illegible]
[illegible]


[illegible]

[illegible]

[illegible]
    [illegible]
    [illegible]
[illegible]


[illegible]

[illegible]
{
    TRACE("Entering CEx15bDoc constructor\n");
#ifdef _DEBUG
    afxDump.SetDepth(1); // Ensure dump of list elements
#endif // _DEBUG
}

[illegible]
{
}

[illegible]
{
    TRACE("Entering CEx15bDoc::OnNewDocument\n");
    [illegible]
        [illegible]

    [illegible]
    [illegible]

    [illegible]
}
```

```
[illegible]

[illegible]
{
    [illegible]
    [illegible]
        [illegible]
    [illegible]
    [illegible]
    [illegible]
        [illegible]
    
}


[illegible]

[illegible]
[illegible]
{
    [illegible]
}

[illegible]
{
    CDocument::Dump(dc);
    dc << "\n" << m_studentList << "\n";
}
[illegible]


[illegible]

[illegible]
{
#ifdef _DEBUG
    Dump(afxDump);
#endif
    while (m_studentList.GetHeadPosition()) {
        delete m_studentList.RemoveHead();
    }
}

[illegible]
{
    DeleteContents();
    UpdateAllViews(NULL);
}
```

см. след. стр.

```
void CEx15bDoc::OnUpdateEditClearall(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(!m_studentList.IsEmpty());
}
```

Обработчики сообщений в *CEx15bDoc*

Команду Clear All (меню Edit) обрабатывает класс документа. В окне Properties утилиты Class View добавлены обработчики сообщений:

Идентификатор объекта	Сообщение	Имя функции-члена
ID_EDIT_CLEAR_ALL	*ON_COMMAND*	*OnEditClearall*
ID_EDIT_CLEAR_ALL	*ON_UPDATE_COMMAND_UI*	*OnUpdateEditClearall*

Переменные-члены

Класс документа содержит внедренный объект *CStudentList* — переменную-член *m_studentList*, хранящую указатели на объекты *CStudent*. Объект-список конструируется при создании объекта *CStudentDoc* и уничтожается при закрытии программы. *CStudentList* определен с помощью *typedef* как *CTypedPtrList* для указателей на *CStudent*.

Конструктор

Конструктор документа устанавливает глубину диагностического вывода, достаточную для вывода информации по отдельным элементам.

GetList

Подставляемая в строку функция *GetList* помогает изолировать представление от документа. Класс документа зависит от типа объектов в списке, в данном случае — от объектов класса *CStudent*. Однако базовый класс «вид», чтобы получить указатель на список, не зная имени его объекта, может вызвать соответствующую функцию-член.

DeleteContents

DeleteContents — переопределенная виртуальная функция, вызываемая другими функциями класса «документ» и каркасом приложений. Ее задача — извлекать из списка указатели на объекты *CStudent* и удалять эти объекты. Важно помнить, что объекты-документы SDI повторно используются после закрытия. *DeleteContents* попутно выводит сведения о списке студентов.

Dump

MFC Application Wizard генерирует заготовку этой функции между строками *#ifdef _DEBUG* и *#endif*. Поскольку глубина диагностического вывода для *afxDump* установлена в конструкторе документа равной 1, выводятся присутствующие в списке объекты *CStudent*.

CEx15bView

Код класса *CEx15bView* показан в листинге.

Ex15bView.h

```
[illegible]
[illegible]

[illegible]

[illegible]
[illegible]
[illegible]
    POSITION    m_position; // текущее положение в списке документов
    CStudentList* m_pList; // копируется из документа
[illegible]
    [illegible]
    [illegible]

[illegible]
    [illegible]

[illegible]
[illegible]
    [illegible]

[illegible]
[illegible]

[illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]

[illegible]
[illegible]
    [illegible]
[illegible]
    [illegible]
    [illegible]
[illegible]

[illegible]
    virtual void ClearEntry();
    virtual void InsertEntry(POSITION position);
    virtual void GetEntry(POSITION position);
    [illegible]
```

см. след. стр.

```
[illegible]
```

Ex15bView.cpp

```
[illegible]
```

```
    ON_COMMAND(ID_STUDENT_HOME, OnStudentHome)
    ON_COMMAND(ID_STUDENT_DELETE, OnStudentDelete)
    ON_COMMAND(ID_STUDENT_END, OnStudentEnd)
    ON_COMMAND(ID_STUDENT_INSERT, OnStudentInsert)
    ON_COMMAND(ID_STUDENT_NEXT, OnStudentNext)
    ON_COMMAND(ID_STUDENT_PREVIOUS, OnStudentPrevious)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_HOME, OnUpdateStudentHome)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_DELETE, OnUpdateStudentDelete)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_END, OnUpdateStudentEnd)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_NEXT, OnUpdateStudentNext)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_PREVIOUS, OnUpdateStudentPrevious)
    ON_BN_CLICKED(IDC_CLEAR, OnClear)
END_MESSAGE_MAP()

// CEx15bView construction/destruction

CEx15bView::CEx15bView()
      : CFormView(CEx15bView::IDD)
    , m_nGrade(0)
    , m_strName(_T(""))
    , m_position(NULL)
{
    TRACE("Entering CEx15bView constructor\n");
}

CEx15bView::~CEx15bView()
{
}

void CEx15bView::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_GRADE, m_nGrade);
    DDX_Text(pDX, IDC_NAME, m_strName);
}

BOOL CEx15bView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CFormView::PreCreateWindow(cs);
}

void CEx15bView::OnInitialUpdate()
{
    TRACE("Entering CEx15bView::OnInitialUpdate\n");
    m_pList = GetDocument()->GetList();
    CFormView::OnInitialUpdate();
```

см. след. стр.

```
}

[illegible]

[illegible]
[illegible]
{
   [illegible]
}

[illegible]
{
   [illegible]
}

[illegible]
{
   [illegible]
   [illegible]
}
[illegible]


[illegible]

[illegible]
{
   TRACE("Entering CEx15bView::OnStudentHome\n");
   // надо проверять, не пуст ли список
   if (!m_pList->IsEmpty()) {
      m_position = m_pList->GetHeadPosition();
      GetEntry(m_position);
   }
}

[illegible]
{
   // вызывается в момент простоя и при вызове меню Student
   POSITION pos;

   // активизирует кнопку, если список не пуст...
   // и текущее положение не совпадает с началом списка
   pos = m_pList->GetHeadPosition();
   pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}

[illegible]
{
```

```
    // удаляет текущий элемент и позиционирует список...
    // на следующий элемент или на начало
    POSITION pos;
    TRACE("Entering CEx15bView::OnStudentDelete\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos == NULL) {
            pos = m_pList->GetHeadPosition();
            TRACE("GetHeadPos = %ld\n", pos);
            if (pos == m_position) {
                pos = NULL;
            }
        }
        GetEntry(pos);
        CStudent* ps = m_pList->GetAt(m_position);
        m_pList->RemoveAt(m_position);
        delete ps;
        m_position = pos;
        GetDocument()->SetModifiedFlag();
        GetDocument()->UpdateAllViews(this);
    }
}

[illegible]
{
    // вызывается в момент простоя и при вызове меню Student
    pCmdUI->Enable(m_position != NULL);
}

[illegible]
{
    TRACE("Entering CEx15bView::OnStudentEnd\n");
    if (!m_pList->IsEmpty()) {
        m_position = m_pList->GetTailPosition();
        GetEntry(m_position);
    }
}

[illegible]
{
    // вызывается в момент простоя и при вызове меню Student
    POSITION pos;

    // активизирует кнопку, если список не пуст...
    // и текущее положение не совпадает с концом списка
    pos = m_pList->GetTailPosition();
    pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}
```

см. след. стр.

```
[illegible]
{
    TRACE("Entering CEx15bView::OnStudentInsert\n");
    InsertEntry(m_position);
    GetDocument()->SetModifiedFlag();
    GetDocument()->UpdateAllViews(this);
}

[illegible]
{
    POSITION pos;
    TRACE("Entering CEx15bView::OnStudentNext\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos) {
            GetEntry(pos);
            m_position = pos;
        }
    }
}

[illegible]
{
    OnUpdateStudentEnd(pCmdUI);
}

[illegible]
{
    POSITION pos;
    TRACE("Entering CEx15bView::OnStudentPrevious\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetPrev(pos);
        if (pos) {
            GetEntry(pos);
            m_position = pos;
        }
    }
}

[illegible]
{
    OnUpdateStudentHome(pCmdUI);
}

[illegible]
    [illegible]
{
    // Вызывается функциями OnInitialUpdate и UpdateAllViews
    TRACE("Entering CEx15bView::OnUpdate\n");
```

```
    m_position = m_pList->GetHeadPosition();
    GetEntry(m_position); // начальные данные для окна представления
}

[illegible]
{
    m_strName = "";
    m_nGrade = 0;
    UpdateData(FALSE);
    ((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
}

[illegible]
{
    if (position) {
        CStudent* pStudent = m_pList->GetAt(position);
        m_strName = pStudent->m_strName;
        m_nGrade = pStudent->m_nGrade;
    }
    else {

        ClearEntry();
    }
    UpdateData(FALSE);
}

[illegible]
{
    if (UpdateData(TRUE)) {
        // UpdateData возвращает FALSE, обнаружив ошибку пользователя
        CStudent* pStudent = new CStudent;
        pStudent->m_strName = m_strName;
        pStudent->m_nGrade = m_nGrade;
        m_position = m_pList->InsertAfter(m_position, pStudent);
    }
}


[illegible]
{
    TRACE("Entering CEx15bView::OnBnClickedClear\n");
    ClearEntry();
}
```

Обработчики сообщений класса *CEx15bView*

В окне Properties утилиты Class View в классе *CEx15bView* создан обработчик уведомляющего сообщения от командной кнопки Clear.

Идентификатор объекта	Сообщение	Имя функции-члена
IDC_CLEAR	*BN_CLICKED*	*OnBnClickedClear*

Так как класс *CEx15bView* произведен от *CFormView*, Class View поддерживает определение переменных-членов диалогового окна. Следующие переменные добавлены мастером Add Member Variable Wizard.

Идентификатор элемента управления	Имя переменной	Категория	Тип переменной
IDC_GRADE	*m_nGrade*	*Value*	*int*
IDC_NAME	*m_strName*	Value	*CString*

Присвойте *m_nGrade* минимальное значение 0, максимальное — 100. Установите предельную длину *m_strName* в 20 символов.

В окне Properties утилиты Class View сопоставляются обработчики командам кнопок. Ниже приведена таблица команд и их обработчиков.

Идентификатор элемента управления	Сообщение	Функция-обработчик
ID_STUDENT_HOME	*COMMAND*	*OnStudentHome*
ID_STUDENT_END	*COMMAND*	*OnStudentEnd*
ID_STUDENT_PREVIOUS	*COMMAND*	*OnStudentPrevious*
ID_STUDENT_NEXT	*COMMAND*	*OnStudentNext*
ID_STUDENT_INSERT	*COMMAND*	*OnStudentInsert*
ID_STUDENT_DELETE	*COMMAND*	*OnStudentDelete*

В каждый обработчик встроен контроль ошибок.

Следующие обработчики сообщений обновления пользовательского интерфейса вызываются или в момент простоя — для обновления состояния кнопок панели инструментов, или при вызове меню Student — для обновления пунктов меню.

Идентификатор элемента управления	Сообщение	Функция-обработчик
ID_STUDENT_HOME	*UPDATE_COMMAND_UI*	*OnUpdateStudentHome*
ID_STUDENT_END	*UPDATE_COMMAND_UI*	*OnUpdateStudentEnd*
ID_STUDENT_PREVIOUS	*UPDATE_COMMAND_UI*	*OnUpdateStudentPrevious*
ID_STUDENT_NEXT	*UPDATE_COMMAND_UI*	*OnUpdateStudentNext*
ID_STUDENT_DELETE	*UPDATE_COMMAND_UI*	*OnUpdateCommandDelete*

Например, кнопка выбора первой записи:



отключена, если список пуст, а также если переменная *m_position* уже указывает на первую запись. Кнопка Previous (предыдущий) отключается в тех же случаях, поэтому она использует тот же обработчик команды обновления пользовательского интерфейса. Поэтому же применяют один обработчик кнопки End (конец) и Next (следующий). Так как вызов функций обновления командного интерфейса иногда происходит с задержкой, обработчики командных сообщений должны содержать проверку ошибок.

Переменные-члены

Переменная *m_position* — что-то вроде курсора для набора объектов в документе. Он ссылается на отображаемый в данный момент объект *CStudent*. Переменная *m_pList* обеспечивает быстрый доступ к списку студентов в документе.

OnInitialUpdate

Виртуальная функция *OnInitialUpdate*, вызываемая при запуске приложения, инициализирует *m_pList* для последующего доступа к списку в документе.

OnUpdate

Виртуальную функцию *OnUpdate* вызывает как функция *OnInitialUpdate*, так и *CDocument::UpdateAllViews*. *OnUpdate* устанавливает текущее положение на начало списка и отображает его первый элемент. В данном случае функция *UpdateAllViews* вызывается только по команде Clear All (меню Edit). В приложении с несколькими представлениями данных в ответ на обновление документа из другого окна представления может понадобиться иная стратегия установки переменной *m_position* в классе *CEx15bView*.

Защищенные виртуальные функции

Перечисленные ниже функции — защищенные виртуальные функции, специально предназначенные для работы с объектами *CStudent*: *GetEntry*, *InsertEntry* и *ClearEntry*. Чтобы выделить в базовый класс универсальные средства обработки списков, эти функции надо переместить в производный класс.

Тестирование программы Ex15b

Заполните поля ввода и, чтобы вставить запись в список, нажмите кнопку:



Повторите эту операцию еще несколько раз, стирая предыдущие данные из полей ввода щелчком кнопки Clear. Закрыв приложение, вы должны увидеть в окне вывода отладочной информации примерно такую запись:

```
 a CEx15bDoc at $4116D0
m_strTitle = Untitled
m_strPathName =
m_bModified = 1
m_pDocTemplate = $4113F1

a CObList at $411624
with 4 elements
   a CStudent at $412770
m_strName = Fisher, Lon
m_nGrade = 67
   a CStudent at $412E80
m_strName = Meyers, Lisa
m_nGrade = 80
   a CStudent at $412880
```

```
m_strName = Seghers, John
m_nGrade = 92
   a CStudent at $4128F0
m_strName = Anderson, Bob
m_nGrade = 87
```

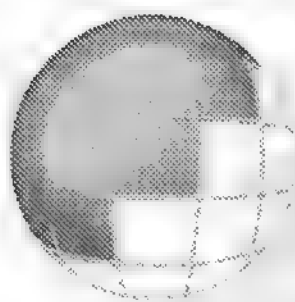
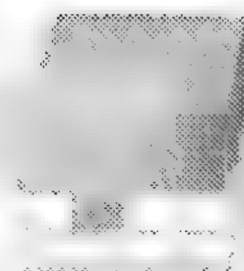
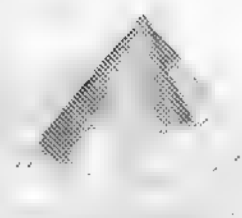
Пара упражнений для читателя

Возможно, вы заметили, что на панели инструментов нет кнопки, позволяющей модифицировать запись о студенте. Попробуйте сами добавить ее и обработчики сообщений. Самой сложной задачей должно быть создание изображения для кнопки.

Вспомните, что класс *CEx15bView* почти готов к тому, чтобы стать универсальным базовым. Попытайтесь выделить виртуальные функции с кодом, характерным для *CStudent*, в производный класс. А потом создайте другой производный класс, который будет использовать новый класс элементов списка, отличный от *CStudent*.

ГЛАВА

16

Чтение и запись документов

Вероятно, вы обратили внимание, что в каждой программе, сгенерированной MFC Application Wizard, есть знакомое меню File с командами New, Open, Save и Save As. В этой главе вы узнаете, как заставить приложение реагировать на них, т. е. считывать и записывать документы.

Вы познакомитесь как с SDI- (Single Document Interface), так и с MDI-программами (Multiple Document Interface). Мы углубимся в теорию каркаса приложений, в том числе рассмотрим множество вспомогательных классов, о которых до сих пор не было сказано ни слова. Путь предстоит трудный, но поверьте: это действительно нужно, иначе вы не сумеете эффективно использовать потенциал каркаса приложений.

В этой главе три программы-примера: SDI-приложение, MDI-приложение на основе программы Ex15b из предыдущей главы и MTI-приложение (Multiple Top-Level Interface). Во всех есть документ со списком студентов и вид, производный от класса *CFormView*. Теперь список студентов можно будет сохранять на диске и считывать с него, применяя сериализацию.

Понятие сериализации

В мире объектно-ориентированного программирования существуют *постоянные* (persistent) объекты, которые можно сохранять на диске при завершении программы и восстанавливать при следующем запуске. Такой процесс сохранения и восстановления объектов называется *сериализацией* (serialization). Поддерживающие ее классы библиотеки MFC содержат функцию-член *Serialize*. Когда каркас приложений вызывает ее, скажем, для объекта класса *CStudent*, данные о студенте либо сохраняются на диске, либо считываются с него.

В библиотеке MFC сериализация не заменяет систему управления базами данных. Все объекты, связанные с конкретным документом, *последовательно* считы-

ваются или записываются в один дисковый файл. Доступ к отдельным объектам в файле по произвольным адресам невозможен. Если вашей программе нужны средства управления базами данных, подумайте об использовании средств доступа к базам данных в MFC и ATL (Active Template Library).

Дисковые файлы и архивы

Как узнать, что должна делать функция *Serialize*: считывать или записывать данные? Как она связывается с дисковым файлом? В MFC-библиотеке дисковые файлы представляются объектами класса *CFile*. Объект *CFile* инкапсулирует описатель двоичного файла, возвращаемый Win32-функцией *CreateFile*. Это *не* указатель на структуру *FILE* буферизованного файла, возвращаемый функцией *fopen* стандартной C-библиотеки, а именно описатель двоичного файла. Каркас приложений использует его при вызовах Win32-функций *ReadFile*, *WriteFile* и *SetFilePointer*.

Если программа не выполняет прямые операции ввода/вывода на диск, а полагается на сериализацию, явной работы с объектами *CFile* можно избежать. «Между» функцией *Serialize* и объектом *CFile* располагается объект-архив класса *CArchive* (рис. 16-1). Он буферизует данные для объекта *CFile* и поддерживает внутренний флаг, указывающий, записывается или считывается архив с диска. С каждым файлом в каждый момент времени связывается только один активный архив. За создание объектов *CFile* и *CArchive*, открытие дискового файла для объекта *CFile* и привязку объекта-архива к файлу отвечает каркас приложений. Все, что вам остается сделать в своей функции *Serialize*, — сохранить/загрузить данные в/из объекта-архива. Каркас приложений вызывает функцию *Serialize* класса документа при обработке команд Open и Save из меню File.

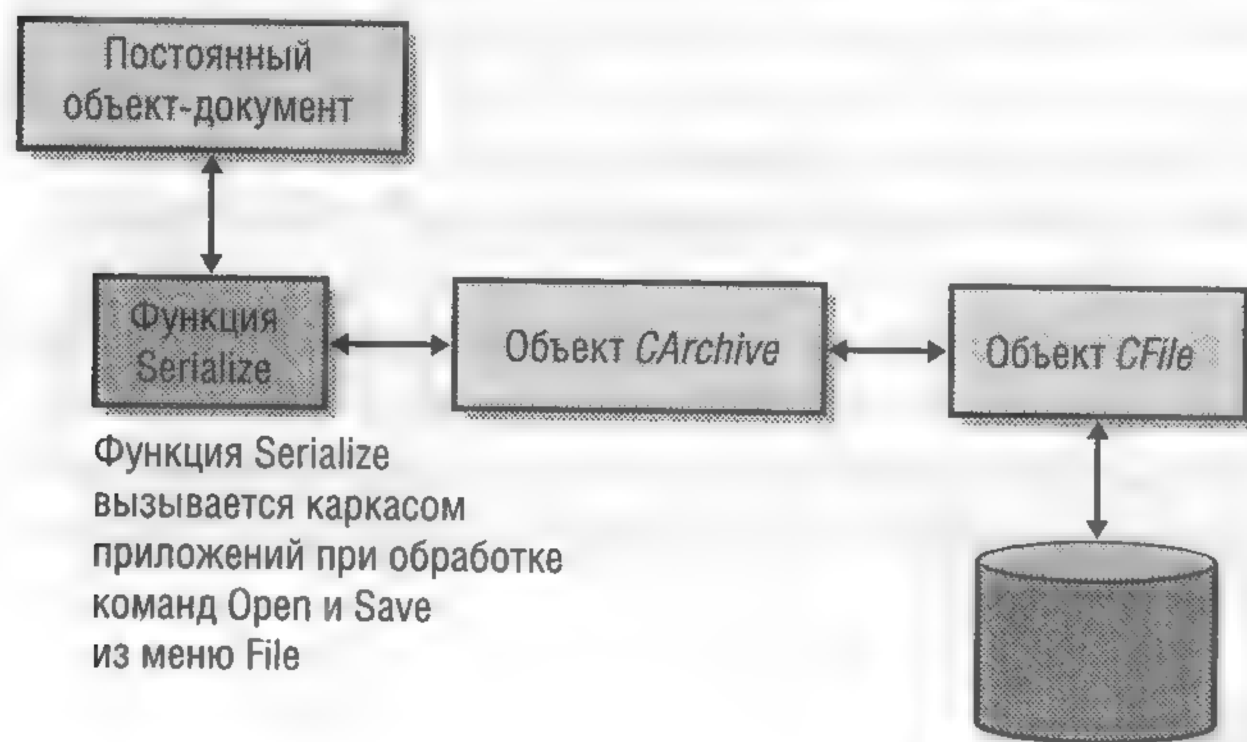


Рис. 16-1. *Процесс сериализации*

Создание сериализуемого класса

Чтобы стать сериализуемым, класс должен быть производным (прямо или косвенно) от *CObject*. Кроме того, в объявлении класса должен присутствовать макрос *DECLARE_SERIAL*, а в файле реализации — макрос *IMPLEMENT_SERIAL*. (Описание этих макросов см. в «MFC Library Reference».) Мы внесем эти макросы в класс *CStudent* из главы 15 и будем использовать его в следующих примерах.

Создание функции *Serialize*

В главе 15 мы создали класс *CStudent*, производный от *CObject*, с такими переменными-членами:

```
public:
   CString   m_strName;
   int    m_nGrade;
```

Теперь напишем для класса *CStudent* функцию *Serialize*. Так как это виртуальная функция класса *CObject*, типы ее параметров и возвращаемого значения должны совпадать с объявленными в *CObject*. Вот функция *Serialize* для класса *CStudent*:

```
void CStudent::Serialize(CArchive& ar)
{
   TRACE("Entering CStudent::Serialize\n ")
   if (ar.IsStoring()) {
      ar << m_strName << m_nGrade;
   }
   else {
      ar >> m_strName >> m_nGrade;
   }
}
```

Обычно функции сериализации вызывают *Serialize* соответствующего базового класса. Если бы *CStudent* был производным, скажем, от *CPerson*, первая строка в ней выглядела бы так:

```
CPerson::Serialize(ar);
```

Функции *Serialize* для класса *CObject* (и *CDocument*, если он не переопределяет эту функцию) не делают ничего полезного, так что вызывать их нет смысла.

Заметьте: *ar* — это параметр, ссылающийся на объект «архив» приложения. Функция-член *CArchive::IsStoring* сообщает, как архив используется в настоящий момент: для записи или считывания. У класса *CArchive* перегружены операторы вставки (<<) и извлечения (>>) для многих встроенных типов C++ (табл. 16-1).

Табл. 16-1. Типы, поддерживаемые операторами вставки (<<) и извлечения (>>)

Тип данных	Описание
BYTE	8 разрядов, беззнаковый
WORD	16 разрядов, беззнаковый
LONG	32 разряда, знаковый
DWORD	32 разряда, беззнаковый
Float	32 разряда
Double	64 разряда, стандарт IEEE
Int	32 разряда, знаковый
Short	16 разрядов, знаковый
Char	8 разрядов, знаковый
Unsigned	32 разряда, беззнаковый

Операторы вставки перегружены для приема параметров по значению, а операторы извлечения — по ссылке. Иногда приходится прибегать к преобразованию типов, чтобы избежать ошибок компиляции. Допустим, имеется переменная-член *m_nType* перечислимого типа. Вот какой код надо написать:

```
ar << (int) m_nType;
ar >> (int&) m_nType;
```

У таких MFC-классов, как *CString* и *CRect*, не являющихся производными от *CObject*, есть свои перегруженные операторы вставки и извлечения для *CArchive*.

Загрузка из архива: внедренные объекты и указатели

Допустим, в объект *CStudent* внедрены другие объекты, не являющиеся экземплярами стандартного класса вроде *CString*, *CSize* или *CRect*. Добавим к классу *CStudent* новую переменную-член:

```
public:
    CTranscript m_transcript;
```

Будем считать, что *CTranscript* — некий нестандартный класс (производный от *CObject*), у которого есть собственная функция *Serialize*. Для *CObject* не предусмотрены перегруженные операторы << и >>, поэтому функция *CStudent::Serialize* приобретает вид:

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
        ar << m_strName << m_nGrade;
    }
    else {
        ar >> m_strName >> m_nGrade;
    }
    m_transcript.Serialize(ar);
}
```

Прежде чем вызывать функцию *CStudent::Serialize* для загрузки из архива записи о студенте, надо создать объект *CStudent*. Внедренный объект *m_transcript* класса *CTranscript* создается вместе с объектом *CStudent* перед вызовом функции *CTranscript::Serialize*. Последняя может загрузить из архива соответствующие данные во внедренный объект *m_transcript*. Запомните одно правило: для внедренных объектов классов, производных от *CObject*, функция *Serialize* вызывается явно.

Теперь допустим, что *CStudent* вместо внедренного объекта содержит *указатель* на *CTranscript*:

```
public:
    CTranscript* m_pTranscript;
```

Можно было бы вызвать функцию *Serialize*, как показано ниже, но при этом пришлось бы самостоятельно создавать объект *CTranscript*:

```
void CStudent::Serialize(CArchive& ar)
{
```

```
    if (ar.IsStoring()) {
        ar << m_strName << m_nGrade;
    }
    else {
        m_pTranscript = new CTranscript;
        ar >> m_strName >> m_nGrade;
    }
    m_pTranscript->Serialize(ar);
}
```

Поскольку операторы вставки и извлечения в *CArchive* для *указателей* на *CObject* на самом деле перегружены, можно написать функцию *Serialize* и так:

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) {
        ar << m_strName << m_nGrade << m_pTranscript;
    }
    else {
        ar >> m_strName >> m_nGrade >> m_pTranscript;
    }
}
```

Для создания объекта *CTranscript* при загрузке данных из архива служат макросы *DECLARE_SERIAL* и *IMPLEMENT_SERIAL* из класса *CTranscript*. Когда объект *CTranscript* записывается в архив, эти макросы заботятся о занесении туда вместе с данными и имени класса. При загрузке архива считывается имя класса, и динамически создается объект нужного класса под управлением сгенерированного макросами кода. После того как объект *CTranscript* сформирован, можно вызвать переопределенную для класса *CTranscript* функцию *Serialize*, чтобы считать данные из дискового файла.

И последнее. Указатель на *CTranscript* сохраняется в переменной-члене *m_pTranscript*. Чтобы избежать «утечки памяти», убедитесь, что в *m_pTranscript* еще не занесен указатель на объект *CTranscript*. Если объект *CStudent* только что создан (а не загружен из архива), указатель на *CTranscript* будет нулевым.

Операторы вставки и извлечения *не работают* с внедренными объектами классов, производных от *CObject*:

```
ar >> m_strName >> m_nGrade >> &m_transcript; // никогда так не делайте
```

Сериализация наборов

Так как все классы наборов — производные от *CObject* и в их объявлениях присутствует вызов макроса *DECLARE_SERIAL*, для сериализации наборов достаточно просто вызвать функцию *Serialize* соответствующего класса набора. В частности, вызов *Serialize* для набора *CObList* объектов *CStudent* инициирует вызов функции *Serialize* для каждого объекта *CStudent*. Но при этом не забывайте об особенностях процесса загрузки наборов из архива:

- если набор содержит указатели на объекты разных классов, каждый из которых происходит от *CObject*, имена этих классов сохраняются в архиве, чтобы

потом можно было корректно восстановить объекты конструктором соответствующего класса;

- если объект-контейнер (скажем, документ) содержит внедренный набор, загружаемые данные добавляются к текущему содержимому набора, поэтому не исключено, что перед загрузкой из архива вам придется очистить такой набор; обычно это делается в виртуальной функции *DeleteContents*, вызываемой каркасом приложений;
- когда из архива загружается набор указателей на *CObject*, над каждым объектом в наборе выполняются следующие операции:
  - определяется класс объекта;
  - для объекта выделяется память из кучи;
  - в выделенную память загружаются данные объекта;
  - указатель на новый объект сохраняется в наборе.

Сериализация внедренного набора объектов *CStudent* описана в программе Ex16a.

Функция *Serialize* и каркас приложений

Итак, теперь вы знаете, как писать функции *Serialize* и что их вызовы могут быть вложенными. Но знаете ли вы, когда вызывается первая функция *Serialize* для запуска процесса сериализации? В каркасе приложений все связано с документом (с объектом класса, производного от *CDocument*). При выборе команды Save или Open из меню File каркас приложений создает объект *CArchive* (и соответствующий объект *CFile*), после чего вызывает функцию *Serialize* из вашего класса документа, передавая ссылку на объект *CArchive*. Затем функция *Serialize* производного класса документа выполняет сериализацию всех его постоянных (не временных) переменных-членов.

Примечание Присмотревшись к какому-нибудь классу документа, сгенерированному MFC Application Wizard, вы заметите, что вместо макросов *DECLARE_SERIAL* и *IMPLEMENT_SERIAL* в нем применяются макросы *DECLARE_DYNCREATE* и *IMPLEMENT_DYNCREATE*. Макросы *SERIAL* не нужны, так как объекты-документы никогда не используются вместе с оператором извлечения *CArchive* и не включаются в наборы; каркас приложений вызывает функцию *Serialize* документа напрямую. Макросы *DECLARE_SERIAL* и *IMPLEMENT_SERIAL* предназначены для остальных «сериализуемых» классов.

SDI-приложение

Вы уже видели целый ряд SDI-приложений с одним классом «документ» и одним классом «вид». В этой главе мы по-прежнему будем работать с единственным классом «вид», но постараемся исследовать взаимосвязи между объектом-приложением, основным окном-рамкой, документом, его представлением, объектом — шаблоном документа и связанными с ними ресурсами строк и меню.

Объект-приложение Windows

Для каждого из ранее рассмотренных приложений MFC Application Wizard автоматически генерировал класс, производный от *CWinApp*, и создавал оператор:

```
CMyApp theApp;
```

Здесь мы видим механизм запуска приложения, создаваемого на базе MFC. Класс *CMyApp* происходит от *CWinApp*, а *theApp* — глобальный экземпляр данного класса, называемый объектом-приложением Windows.

Теперь рассмотрим последовательность операций, выполняемых при запуске Windows-приложения на базе MFC.

1. Windows загружает программу в память.
2. Создается глобальный объект *CWinApp*. (Все глобально объявленные объекты формируются в момент загрузки программы.)
3. Windows вызывает глобальную функцию *WinMain*, которая является частью MFC-библиотеки. (*WinMain* эквивалентна функции *main* приложений текстового режима; обе — главные точки входа в программу.)
4. *WinMain* отыскивает единственный экземпляр класса, производного от *CWinApp*.
5. *WinMain* вызывает для *theApp* функцию-член *InitInstance*, переопределенную в вашем производном классе приложения.
6. Переопределенная функция *InitInstance* инициирует загрузку документа и создание основного окна-рамки и окна представления.
7. *WinMain* вызывает для *theApp* функцию-член *Run*, которая организует распределение оконных и командных сообщений.

Вы можете переопределить и другую важную функцию-член *CWinApp* — *ExitInstance*, вызываемую при завершении приложения после закрытия всех его окон.

Примечание Windows допускает одновременное выполнение нескольких экземпляров программы. Функция *InitInstance* вызывается всякий раз, когда запускается новый экземпляр программы. В Win32 каждый такой экземпляр — отдельный процесс. И то, что на виртуальные адресные пространства каждого процесса проецируется один и тот же код, — лишь совпадение. Если нужно найти другие выполняемые экземпляры программы, вызовите Win32-функцию *FindWindow* или — для связи между процессами — определите общую секцию данных или файл, проецируемый в память.

Класс шаблона документа

Взгляните на функцию *InitInstance*, сгенерированную MFC Application Wizard для вашего производного класса приложения:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CEx16aDoc),
    RUNTIME_CLASS(CMainFrame),    // main SDI frame window
```

```
    RUNTIME_CLASS(CEx16aView));
AddDocTemplate(pDocTemplate);
```

Если вы не собираетесь использовать разделяемые окна или множественные представления данных, то с объектом «шаблон документа» вы встретитесь фактически только в этом месте программы. В данном случае это объект класса *CSingleDocTemplate*, производного от *CDocTemplate*. Класс *CSingleDocTemplate* применяется только в SDI-приложениях, так как в них не бывает нескольких объектов «документ». Что касается *AddDocTemplate*, то это функция-член класса *CWinApp*.

Вызов *AddDocTemplate* (совместно с вызовом конструктора шаблона документа) устанавливает взаимосвязь классов приложения, документа, окна представления и основного окна-рамки. Объект-приложение, конечно, существует и до создания шаблона, но объектов «документ», «рамка» и «вид» в этот момент еще *нет*. Каркас приложений создает их динамически, когда это необходимо.

Такое динамическое создание объектов — пример искусного использования языка C++. Поскольку в определении и реализации класса присутствуют макросы *DECLARE_DYNCREATE* и *IMPLEMENT_DYNCREATE*, библиотека MFC способна создавать объекты класса динамически. Без этого в программу пришлось бы жестко «зашить» намного больше взаимосвязей между классами приложения. Так, в производном классе приложения понадобился бы код для создания документа, его представления и рамки как объектов конкретных производных классов. Это нарушило бы объектно-ориентированную природу программы.

В системе, основанной на шаблонах, нужен лишь макрос *RUNTIME_CLASS*. Заметьте: чтобы макрос работал правильно, надо указать в нем объявление класса.

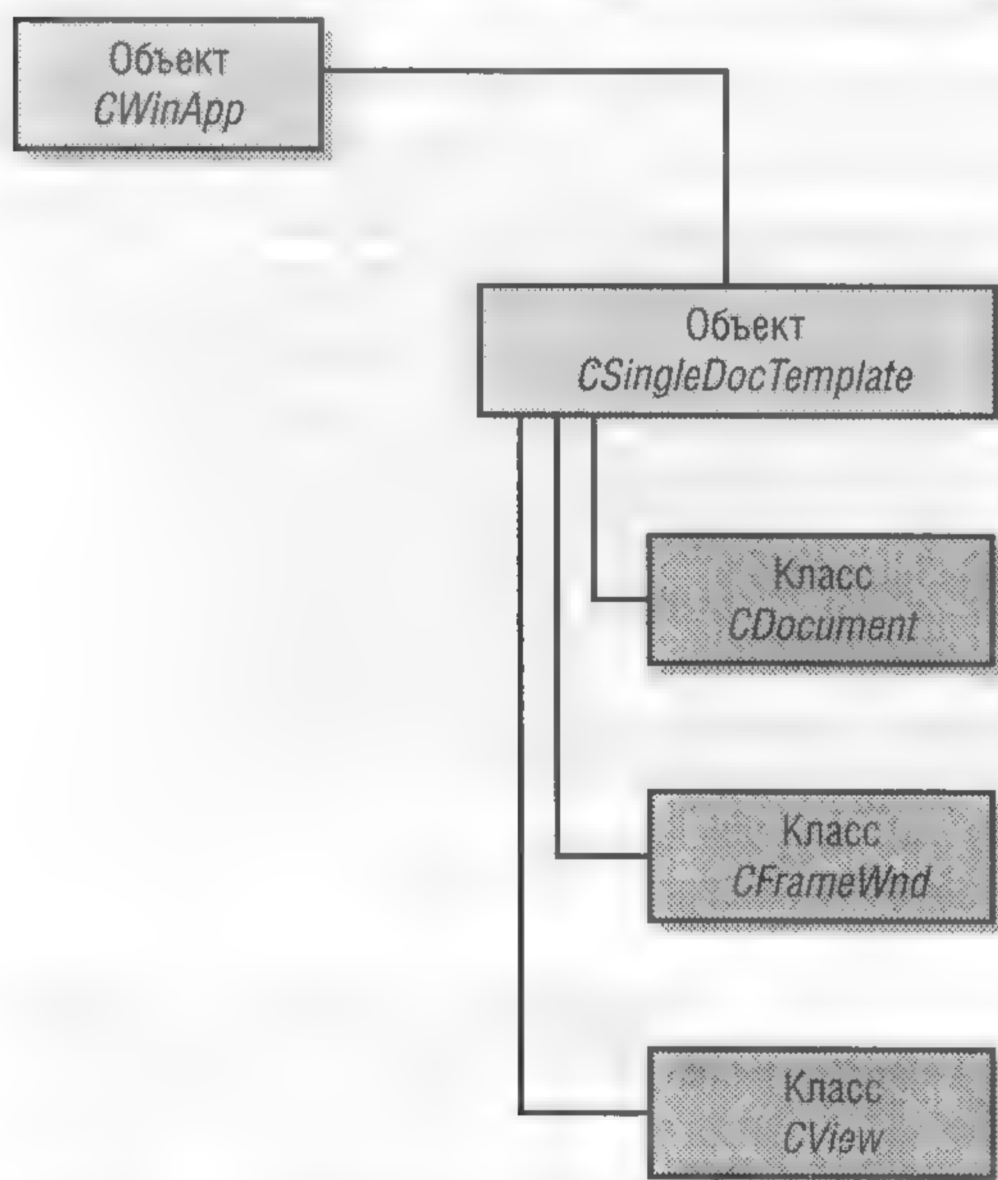


Рис. 16-2. *Взаимосвязи классов*

На рис. 16-2 показаны взаимосвязи классов, а на рис. 16-3 — объектов. У SDI-приложения только один шаблон (и ассоциированные с ним группы классов), а

во время его работы — только один объект документа и один объект основного окна-рамки.

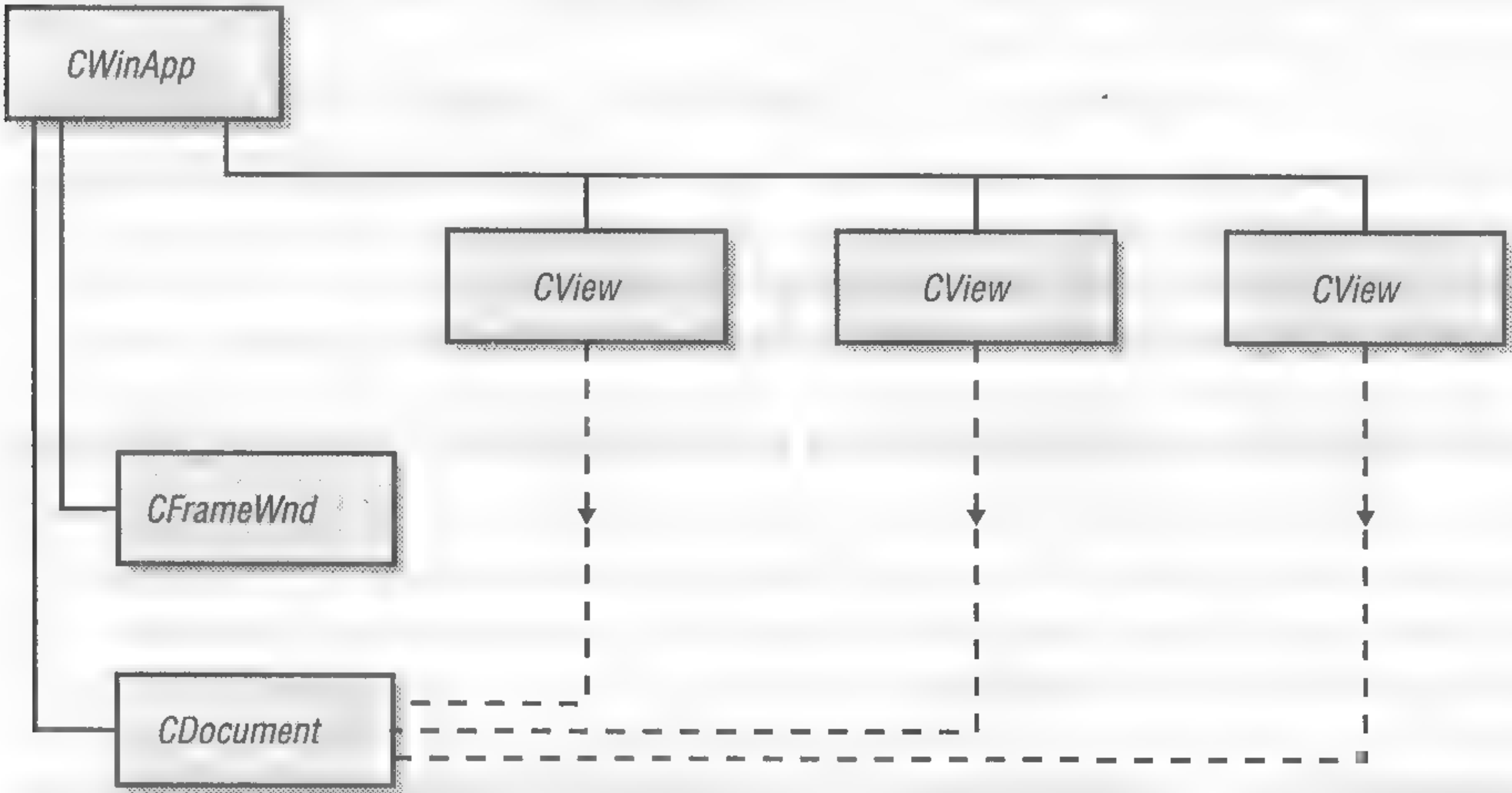


Рис. 16-3. *Взаимосвязи объектов*

Примечание Динамическое создание объектов существовало в MFC-библиотеке еще до появления в языке C++ возможности *идентификации типов в период выполнения* (runtime type identification, RTTI). Однако средства MFC значительно превосходят возможности RTTI, и MFC-библиотека продолжает для динамического создания объектов использовать именно их.

Ресурс шаблона документа

Первый параметр функции *AddDocTemplate* — *IDR_MAINFRAME*, идентификатор строкового ресурса. Вот что MFC Application Wizard генерирует в RC-файле проекта Ex16a:

```
IDR_MAINFRAME
    "Ex16a\n"                  // заголовок окна приложения
    "\n"                       // основа для имени документа по умолчанию
                               // (если не задано, то "Untitled")
    "Ex16a\n"                  // имя типа документа
    "Ex16a Files (*.16a)\n"    // описание типа документа и фильтр
    ".16a\n"                   // расширение документов этого типа
    "Ex16a.Document\n"         // идентификатор типа файла в реестре
    "Ex16a.Document"           // описание типа файла в реестре
```

Примечание Конкатенация (сцепление) строк компилятором ресурсов не поддерживается. Взгляните на файл Ex16a.rc: отдельные «подстроки» собраны в одну длинную строку.

IDR_MAINFRAME определяет одну строку, разбитую на подстроки разделителем строк (\n). Эти подстроки отображаются на экране в тот или иной момент исполнения программы. Строка *16a* — расширение файлов документов по умолчанию, заданное для MFC Application Wizard.

Кроме строк, идентификатор *IDR_MAINFRAME* определяет значок приложения, ресурсы панелей инструментов и меню. Эти ресурсы генерирует MFC Application Wizard, а программист работает с ними через редактор ресурсов.

Итак, вы увидели, как шаблон *AddDocTemplate* связывает воедино все элементы приложения. Но пока не создано ни одного окна, на экране ничего нет.

Множественное представление документа в SDI-программах

Поддержка нескольких представлений документа в SDI-приложении более замысловата. Можно просто определить элемент меню, позволяющий выбрать конкретное представление данных, или же создать несколько представлений в разделяемом окне. Оба способа мы рассмотрим в главе 18.

Создание пустого документа: функция *CWinApp::OnFileNew*

Функция *InitInstance* вашего класса приложения, вызвав функцию-член *AddDocTemplate*, затем вызывает (неявно, через *CWinApp::ProcessShellCommand*) другую важную функцию-член класса *CWinApp* — *OnFileNew*. Последняя, обращаясь к *CWinApp::OpenDocumentFile*, распутывает «паутину» взаимосвязанных имен классов и делает следующее.

1. Создает объект-документ, не пытаясь считать данные с диска.
2. Создает объект основного окна-рамки (класса *CMainFrame*) и основного окна, но не отображает их на экране. У основного окна-рамки есть меню *IDR_MAINFRAME*, панель инструментов и строка состояния.
3. Формирует объект «вид» и соответствующее окно, не отображая его на экране.
4. Устанавливает связи между объектами «документ», «основное окно» и «представление». Не путайте связи между объектами со связями между *классами*, установленными вызовом *AddDocTemplate*.
5. Вызывает для объекта «документ» виртуальную функцию-член *CDocument::OnNewDocument*, которая обращается к виртуальной функции *DeleteContents*.
6. Вызывает для объекта «вид» виртуальную функцию-член *CView::OnInitialUpdate*.
7. Вызывает для объекта-рамки виртуальную функцию-член *CFrameWnd::ActivateFrame*, чтобы вывести на экран основное окно-рамку вместе с меню, окном представления, панелью инструментов и строкой состояния.

Примечание Некоторые из перечисленных функций вызываются не из *OpenDocumentFile*, а неявно самим каркасом приложений.

В SDI-приложении объекты «вид», «документ» и «основное окно-рамка» создаются только раз и существуют на протяжении всей жизни программы. Функцию *CWinApp::OnFileNew* вызывает функция *InitInstance*. Кроме того, она вызывается в ответ на выбор в меню File команды New. В данном случае *OnFileNew* должна вести себя иначе. Она не формирует объекты «вид», «документ» и «рамка», так как

они уже созданы. Вместо этого она использует существующие объекты повторно и выполняет операции 5, 6 и 7. Заметьте: *OnFileNew* всегда вызывает (неявно) *DeleteContents* для очистки документа.

Функция *OnNewDocument* класса «документ»

В главе 15 вы уже встречали функцию-член класса «вид» *OnInitialUpdate* и функцию-член *OnNewDocument* класса «документ». Если бы SDI-приложение не использовало объект-документ повторно, *OnNewDocument* была бы не нужна, так как всю инициализацию документа можно было бы провести в конструкторе его класса. Но в реальности вы должны переопределить *OnNewDocument*, чтобы инициализировать объект-документ всякий раз, когда пользователь выбирает в меню File команду New или Open. MFC Application Wizard поможет вам в этом, создав заготовку функции в сгенерированном производном классе документа.

Примечание Неплохо бы свести к минимуму объем операций, выполняемых в конструкторах. Чем их меньше, тем ниже вероятность сбоя в конструкторе — а такие ошибки могут иметь тяжкие последствия. Функции, подобные *CDocument::OnNewDocument* и *CView::OnInitialUpdate*, — идеальное место для начальной очистки. Если возникнут проблемы, вы сможете вывести сообщения в информационном окне, а при вызове *OnNewDocument* — возвратить *FALSE*. Обе функции можно вызывать для данного объекта неоднократно. Если какие-то действия надо выполнить один раз, объявите специальную переменную-член (флаг) — она послужит признаком «первого вызова».

Связывание File Open с кодом сериализации: функция *OnFileOpen*

Генерируя приложение, MFC Application Wizard сопоставляет команде Open из меню File функцию-член *CWinApp::OnFileOpen*, которая делает следующее.

1. Предлагает пользователю выбрать файл.
2. Вызывает виртуальную функцию-член *CDocument::OnOpenDocument* для существующего объекта-документа. Та открывает файл, вызывает *CDocument::DeleteContents*, создает объект *CArchive*, подготовленный для загрузки, и вызывает функцию *Serialize* документа, которая загружает данные из архива.
3. Вызывает функцию *OnInitialUpdate* класса «вид».

Альтернатива команде Open меню File — *список последних открывавшихся файлов* (Most Recently Used, MRU). Каркас приложений запоминает последние 4 файла и отображает их имена в меню File. В промежутке между запусками программы эти имена хранятся в реестре Windows.

Примечание Можно изменить число запоминаемых файлов, вызвав с соответствующим параметром функцию *LoadStdProfileSetting* в функции *InitInstance* класса приложения.

Функция *DeleteContents* класса «документ»

При загрузке данных из дискового файла в существующий объект-документ SDI надо стереть текущее содержимое объекта. Лучший способ — переопределить виртуальную функцию *CDocument::DeleteContents* в производном классе документа. Как вы видели в главе 15, такая переопределенная функция делает все, что нужно для очистки переменных-членов класса документа. При выборе в меню File команд New и Open функции *OnFileNew* и *OnFileOpen* класса *CDocument* обращаются к *DeleteContents*, а значит, она вызывается сразу после создания объекта-документа (и вновь вызывается при закрытии документа).

Чтобы ваши классы документов работали в SDI-приложениях, очищайте содержимое документа в функции *DeleteContents*, а не в деструкторе. Последний используйте только для очистки элементов, существующих на протяжении всей жизни объекта.

Связывание File Save и File Save As с кодом сериализации

MFC Application Wizard, генерируя приложение, сопоставляет команде Save меню File функцию-член *OnFileSave* класса *CDocument*. Последняя вызывает функцию *OnSaveDocument* класса *CDocument*, которая в свою очередь обращается к функции *Serialize* документа, передавая ей объект-архив, подготовленный для сохранения. Команда Save As из меню File обрабатывается аналогично — ей сопоставляется функция *OnFileSaveAs* класса *CDocument*, которая вызывает *OnSaveDocument*. Все операции с файлами, необходимые для сохранения документа на диске, осуществляет здесь каркас приложений.

Примечание Очевидно, что командам File New и File Open сопоставляются функции-члены класса *приложения*, а File Save и File Save As связываются с функциями-членами класса *документа*. File New связана с *OnFileNew*. Версия *InitInstance* для SDI-приложения тоже вызывает *OnFileNew* (неявно). Объект-документ не существует в момент вызова *InitInstance* каркасом приложений, поэтому *OnFileNew* не может быть функцией-членом *CDocument*. Но при сохранении документа объект-документ, разумеется, существует.

Флаг изменения документа

Многие приложения Windows, ориентированные на документ, отслеживают изменения в нем. При закрытии документа или выходе из программы появляется окно с запросом, сохранить ли текущий документ. Каркас MFC-приложений поддерживает такое поведение при помощи переменной-члена *m_bModified* класса *CDocument*. Эта логическая переменная равна *TRUE*, если документ *изменен* (dirty), и *FALSE*, если нет.

Доступ к защищенному флагу *m_bModified* осуществляется через функции-члены *SetModifiedFlag* и *IsModified* класса *CDocument*. Когда документ создается, открывается или сохраняется на диске, флажок объекта-документа устанавливается в *FALSE*. При изменении его данных нужно устанавливать этот флажок в *TRUE* с помощью *SetModifiedFlag*. Виртуальная функция *CDocument::SaveModified*, которую каркас приложений вызывает, когда пользователь закрывает документ, отображает инфор-

мационное окно, если флажок *m_bModified* установлен в *TRUE*. Если вам нужно выполнить другие действия, переопределите эту функцию.

В примере Ex16a вы увидите, как простейшая функция, обновляющая командный пользовательский интерфейс, с помощью *IsModified* управляет состоянием кнопки на панели инструментов и командой в меню, обеспечивающих доступ к сохранению документа. При изменении документа кнопка на панели инструментов с изображением дискеты активизируется, а когда пользователь сохраняет файл, она блекнет.

Примечание SDI-программы, написанные на базе MFC, ведут себя несколько иначе, нежели другие SDI-приложения для Windows вроде Notepad (Блокнот). Типичная последовательность событий выглядит так:

1. пользователь создает документ и сохраняет его на диске, скажем, как test.dat;
2. пользователь изменяет документ;
3. пользователь выбирает команду Open из меню File и указывает файл test.dat.

При выборе команды File Open программа Notepad спрашивает, сохранить ли изменения в документе, сделанные на этапе 2. Если пользователь отвечает «нет», программа вновь считывает документ с диска. Приложение на MFC считает изменения постоянными, не перезагружая при этом файл.

Пример Ex16a: сериализация в SDI-документе

Программа Ex16a очень похожа на Ex15b. Диалоговое окно для ввода информации о студенте и растровые изображения — те же, класс «вид» тоже не изменился. Но в Ex16a мы добавим сериализацию вместе с функцией обновления командного интерфейса для File Save. Заголовочные файлы и файлы реализации для классов «вид» и «документ» из этого примера будут использованы и в Ex16b.

Далее приведен весь новый код, отличающийся от кода Ex15b, причем выделены все дополнения и изменения по сравнению с кодом, сгенерированным мастерами. Список файлов и классов в Ex16a сведен в табл. 16-2.

Табл. 16-2. Файлы и классы программы Ex16a

Заголовочный файл	Файл с исходным кодом	Класс	Описание
Ex16a.h	Ex16a.cpp	*CEx16aApp*	Класс приложения (создан MFC Application Wizard)
		CAboutDlg	Диалоговое окно About
MainFrm.h	MainFrm.cpp	*CMainFrame*	Основное окно-рамка SDI-приложения
Ex16aDoc.h	Ex16aDoc.cpp	*CEx16aDoc*	Документ с данными о студенте
Ex16aView.h	Ex16aView.cpp	*CEx16aView*	Представление информации о студенте (из Ex15b)
Student.h	Student.cpp	*CStudent*	Запись о студенте
StdAfx.h	StdAfx.cpp		Предкомпилированные заголовочные файлы (с добавлением afxtempl.h)

CStudent

Файл Student.h из Ex16a практически тот же, что и в проекте Ex15b. Заголовочный файл вместо:

```
DECLARE_SERIAL(CStudent)
```

содержит макрос:

```
DECLARE_DYNAMIC(CStudent)
```

а файл реализации вместо:

```
IMPLEMENT_SERIAL(CStudent, CObject, 0)
```

содержит макрос:

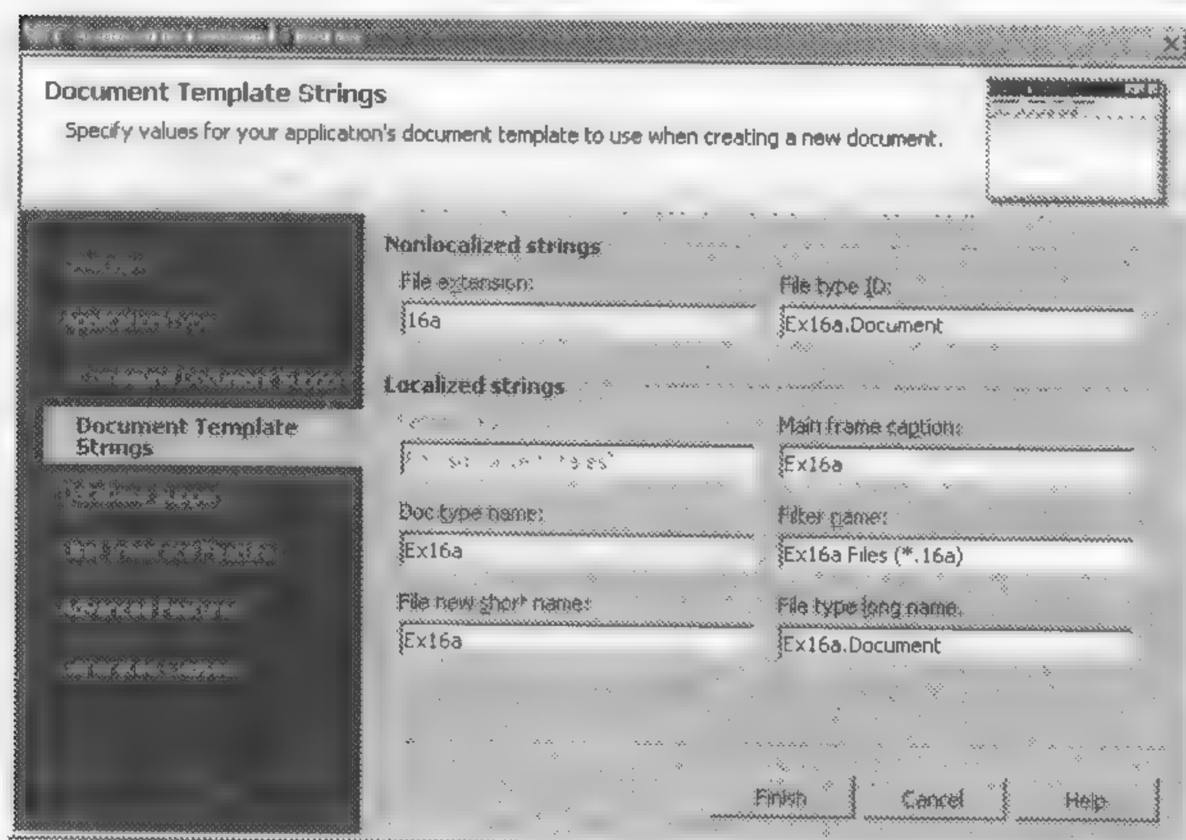
```
IMPLEMENT_DYNAMIC(CStudent, CObject)
```

Кроме того, добавлена виртуальная функция *Serialize*.

CEx16aApp

Файлы класса приложения в этом приложении содержат только код, сгенерированный MFC Application Wizard. Это приложение сгенерировано с расширением файла документа по умолчанию и с поддержкой запуска из Microsoft Windows Explorer (Проводник), а также с поддержкой drag-and-drop. Эти возможности мы обсудим позже в этой главе.

Чтобы сгенерировать дополнительный код, нужно при первом запуске MFC Application Wizard на странице Document Template Strings в File Extension ввести расширение файлов:



Это гарантирует, что строка ресурса шаблона документа содержит правильное расширение, а в функцию-член *InitInstance* класса приложения внесен код поддержки запуска из проводника. Можно изменить и другие подстроки ресурсов.

Ex16a.h

[illegible]

Ex16a.cpp

[illegible]

см. след. стр.

```
// C[illegible]App construction

C[illegible]App::C[illegible]App()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

// The one and only C[illegible]App object
C[illegible]App theApp;

// C[illegible]App initialization

BOOL C[illegible]App::InitInstance()
{
    // InitCommonControlsEx() is required on Windows XP if an application
    // manifest specifies use of ComCtl32.dll version 6 or later to enable
    // visual styles. Otherwise, any window creation will fail.
    InitCommonControls();
    CWinApp::InitInstance();

    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
    AfxEnableControlContainer();
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need
    // Change the registry key under which our settings are stored
    // TODO: You should modify this string to be something appropriate
    // such as the name of your company or organization
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));
    LoadStdProfileSettings(4);  // Load standard INI file options
                                // (including MRU)
    // Register the application's document templates.  Document templates
    //  serve as the connection between documents, frame windows and views
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(C[illegible]Doc),
        RUNTIME_CLASS(CMainFrame),       // main SDI frame window
        RUNTIME_CLASS(C[illegible]View));
    AddDocTemplate(pDocTemplate);
    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    // Parse command line for standard shell commands, DDE, file open
```

CMainFrame

Код класса основного окна-рамки практически не изменен по сравнению с кодом, сгенерированным MFC Application Wizard. Переопределенная функция *ActivateFrame* и обработчик сообщения *WM_DROPFILES* присутствуют только для трассировки.

MainFrm.h

```
[illegible]
public:
    afx_msg void OnDropFiles(HDROP hDropInfo);
    virtual void ActivateFrame(int nCmdShow = -1);
};
```

MainFrm.cpp

```
[illegible]
```

```
[illegible]
```

см. след. стр.

13—2064

```
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
    return 0;
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    [illegible]
    cs.style ... CREATESTRUCT ...
    return TRUE;
}

// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif //_DEBUG
// CMainFrame message handlers
void CMainFrame::OnEndSession(BOOL bEnding)
{
    TRACE("Entering CMainFrame::OnEndSession\n");
    CFrameWnd::OnEndSession(bEnding);
}
void CMainFrame::ActivateFrame(int nCmdShow)
{
    TRACE("Entering CMainFrame::ActivateFrame\n");
    CFrameWnd::ActivateFrame(nCmdShow);
}
```

Класс *CEx16aDoc*

Этот класс идентичен классу *CEx15bDoc* из главы 15 за исключением функций *Serialize*, *DeleteContents*, *OnOpenDocument* и *OnUpdateFileSave*.

Serialize

К коду функции, сгенерированному MFC Application Wizard, добавлена всего одна строка сериализации списка студентов:

```
//////////////////////////////////////////////////////////////////////
// CEx16aDoc serialization

void CEx16aDoc::Serialize(CArchive& ar)
{
```

```
    TRACE("Entering CEx16aDoc::Serialize\n");
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
    m_studentList.Serialize(ar);
}
```

DeleteContents

Оператор *Dump* заменен простой директивой *TRACE*. Вот модифицированный код:

```
void CEx16aDoc::DeleteContents()
{
    TRACE("Entering CEx16aDoc::DeleteContents\n");
    while (m_studentList.GetHeadPosition()) {
        delete m_studentList.RemoveHead();
    }
}
```

OnOpenDocument

Эта виртуальная функция переопределена только для того, чтобы вывести трассировочное сообщение:

```
BOOL CEx16aDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    TRACE("Entering CEx16aDoc::OnOpenDocument\n");
     if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;

    // TODO: Add your specialized creation code here

    return TRUE;
}
```

OnUpdateFileSave

Эта функция таблицы сообщений отключает кнопку File Save на панели инструментов, если флаг изменения документа не установлен. Класс «вид» управляет состоянием этого флажка через функцию *SetModifiedFlag*.

```
void CEx16aDoc::OnUpdateFileSave(CCmdUI* pCmdUI)
{
    // отключить кнопку с изображением дискеты, если файл не изменен
    pCmdUI->Enable(IsModified());
}
```

CEx16aView

Код класса *CEx16aView* позаимствован из класса *CEx15bView* (см. главу 15).

Тестирование приложения Ex16a

Собрав программу, запустите ее под управлением отладчика и протестируйте, введя какие-нибудь данные и сохранив их на диске под именем Test.16a. (Расширение *.16a* вводить не обязательно.)

Выйдите из программы, перезапустите ее и откройте сохраненный вами файл. Есть ли там введенная ранее информация? Загляните в окно отладчика и проверьте последовательность вызовов функций — она должна содержать сообщения о чтении и записи документа о студентах при загрузке и сохранении документа.

Запуск программ из Windows Explorer и операция drag-and-drop

В прошлом пользователи «персоналок» запускали какую-то программу, а потом выбирали файл (иногда называемый «документом»), который содержал данные в формате, понятном этой программе. Так работали многие программы для MS-DOS. Старый диспетчер программ Windows сделал запуск программы проще — двойным щелчком ее значка. В то же время пользователям Apple Macintosh работать было еще легче: они дважды щелкали значок документа, а ОС сама определяла, какую программу запустить.

В современном Windows Explorer (Проводник) программу можно запускать двойным щелчком не только ее значка, но и значка одного из ее документов. Но как проводник определяет, какую программу запускать? Для «привязки» документа к программе применяется реестр. В основе сопоставления лежит расширение имени файла, которое мы определяли в MFC Application Wizard. Но запуск — это далеко не все. Установив связь определенного вида документов с конкретной программой, пользователи могут запускать эту программу, дважды щелкнув значок ее документа или перетащив его мышью из проводника на значок программы. Кроме того, значок документа можно переместить на принтер, и программа распечатает этот документ.

Регистрация программы

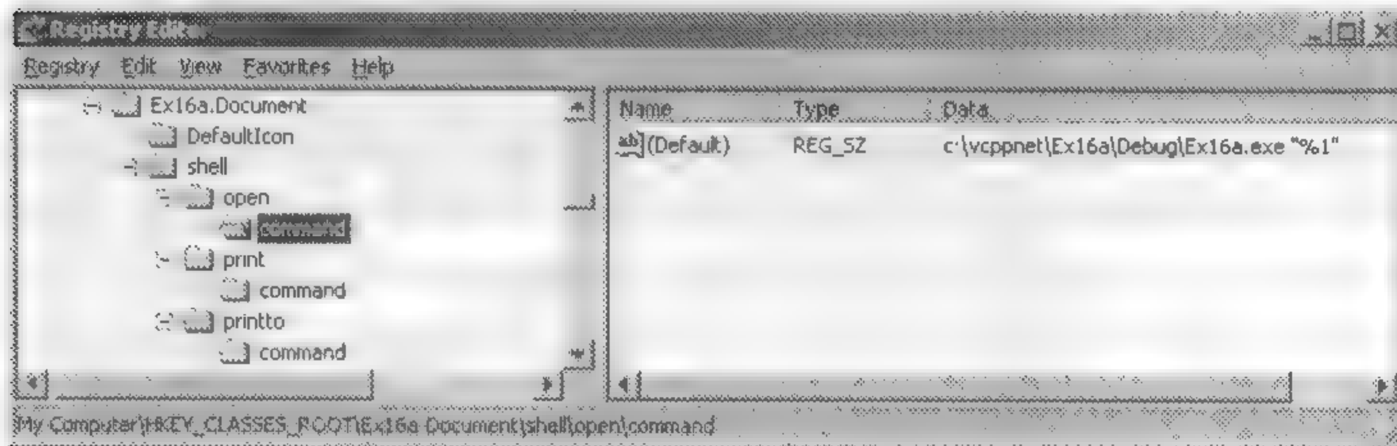
В главе 14 мы выяснили, как сохранять данные в реестре Windows, добавив в функцию *InitInstance* вызов *SetRegistryKey*. Независимо от того, добавили вы этот вызов или нет, ваша программа может при запуске записывать информацию об ассоциациях файлов в другую часть реестра. Для активизации этой возможности укажите расширение имен файлов при создании приложения средствами MFC Application Wizard, который внесет расширение в строку шаблона документа и вставит в функцию *InitInstance* вызов:

```
RegisterSnellFileTypes(True);
```

Теперь программа добавит в реестр два элемента. В разделе HKEY_CLASSES_ROOT она создаст подраздел и строку данных. В Ex16a она выглядит так:

```
.16A = Ex16a.Document
```

Первый элемент — это выбранный для вас мастером MFC Application Wizard идентификатор типа файла, а Ex16a.Document — раздел самой программы. Параметры подраздела Ex16a.Document, также расположенного в разделе HKEY_CLASSES_ROOT, таковы:



В реестре содержится полное имя программы Ex16a с указанием пути. Теперь проводник, используя реестр, может перейти от расширения к идентификатору типа файла и к самой программе. После того как расширение зарегистрировано, проводник находит значок документа и отображает его рядом с именем файла.

Двойной щелчок документа

Когда пользователь дважды щелкает значок документа, проводник запускает соответствующую SDI-программу, передавая ей в командной строке имя выбранного файла. MFC Application Wizard генерирует в функции *InitInstance* вызов *EnableShellOpen*, что обеспечивает поддержку запуска посредством DDE-сообщения. Эта технология применялась в File Manager Windows NT 3.51. Windows Explorer умеет запускать SDI-приложение и без этого вызова.

Активизация механизма drag-and-drop

Чтобы запущенная программа могла открывать файлы, которые пользователь перетаскивает в нее из проводника, вызовите функцию *DragAcceptFiles* класса *CWnd* для основного окна-рамки приложения. Открытая переменная-член *m_pMainWnd* объекта-приложения указывает на объект *CFrameWnd* (или *CMDIFrameWnd*). Когда пользователь «бросает» значок файла внутри окна-рамки, окно получает сообщение *WM_DROP_FILES*, что приводит к вызову обработчика *CFrameWnd::OnDropFiles*. Следующая строка из *InitInstance*, генерируемая MFC Application Wizard, активизирует открытие файлов операцией drag-and-drop:

```
m_pMainWnd->DragAcceptFiles();
```

Параметры запуска программы

При выборе в меню Start команды Run или двойного щелчка значка в проводнике программа запускается без параметров в командной строке. Функция *InitInstance* обрабатывает командную строку, вызывая *ParseCommandLine* и *ProcessShellCommand*. Если в командной строке содержится нечто, напоминающее имя файла, программа сразу загружает этот файл. Поэтому можно создать ярлык Windows, который умеет запускать программу с заданным файлом документа.

Эксперименты с запуском программы из Windows Explorer и операцией drag-and-drop

Собрав Ex16a, попробуйте запустить ее из проводника. Однако сначала запустите программу как обычно, чтобы поместить в реестр начальные записи. Сохраните на диске по крайней мере один файл с расширением *.16a* и закройте программу Ex16a. Запустите проводник и откройте каталог с 16а-файлами. Дважды щелкните один из них в правой панели окна. Ваша программа должна запуститься и автоматически загрузить выбранный файл. Теперь, когда запущены и Ex16a, и проводник, попробуйте перетащить другой файл из Explorer в окно Ex16a. Программа откроет новый файл, как при выборе команды Open из меню File.

Возможно, вы захотите просмотреть записи в реестре Windows, относящиеся к Ex16a. Тогда запустите программу Regedit (Regedt32 в Windows 2000/XP) и раскройте раздел HKEY_CLASSES_ROOT. Просмотрите содержимое подразделов .16A и Ex16a.Document. Кроме того, раскройте раздел HKEY_CURRENT_USER (или HKEY_USERS) и изучите подраздел Software. Там в разделе Ex16a вы должны найти подраздел Recent File List (список последних открывавшихся файлов). Программа Ex16a вызывает *SetRegistryKey* со строкой «Local AppWizard-Generated Applications», поэтому подраздел с именем этой программы находится в разделе Ex16a.

Работа с документами в MDI-приложениях

MFC поддерживает не только SDI-приложения, но и MDI-программы. В этой главе вы узнаете, как загружаются и сохраняются файлы документов в MDI-приложениях. По-видимому, именно MDI-приложения лучше программировать при помощи MFC. В частности, MFC Application Wizard по умолчанию предлагает создать MDI-приложение, да и большинство примеров программ, поставляемых с Microsoft Visual C++, — это MDI-приложения.

Вы изучите сходства и различия SDI- и MDI-приложений и научитесь преобразовывать SDI- в MDI-программы. Но приступать к изучению MDI-программ можно, только досконально разобравшись в материалах главы 15.

Прежде чем обратиться к коду MFC-библиотеки для MDI-приложений, приглядимся к работе программ этого типа. Посмотрите на Visual C++ .NET — это MDI-приложение, «множественными документами» которого выступают файлы с исходным текстом программ. Но это не самое типичное MDI-приложение, так как документы в Visual C++ .NET группируются в проекты. Предпочтительнее исследовать Microsoft Word или — еще лучше — MDI-приложение на базе MFC и сгенерированное средствами MFC Application Wizard.

Типичное MDI-приложение в стиле MFC

Пример Ex16b (рис. 16-4) — это MDI-версия программы Ex16a.

Открыты два разных файла документов, каждый в своем дочернем MDI-окне, но активно только одно дочернее окно. У приложения одно меню и одна панель инструментов; все команды маршрутизируются в активное дочернее окно. Заголовок основного окна содержит имя файла документа, открытого в активном дочернем окне.

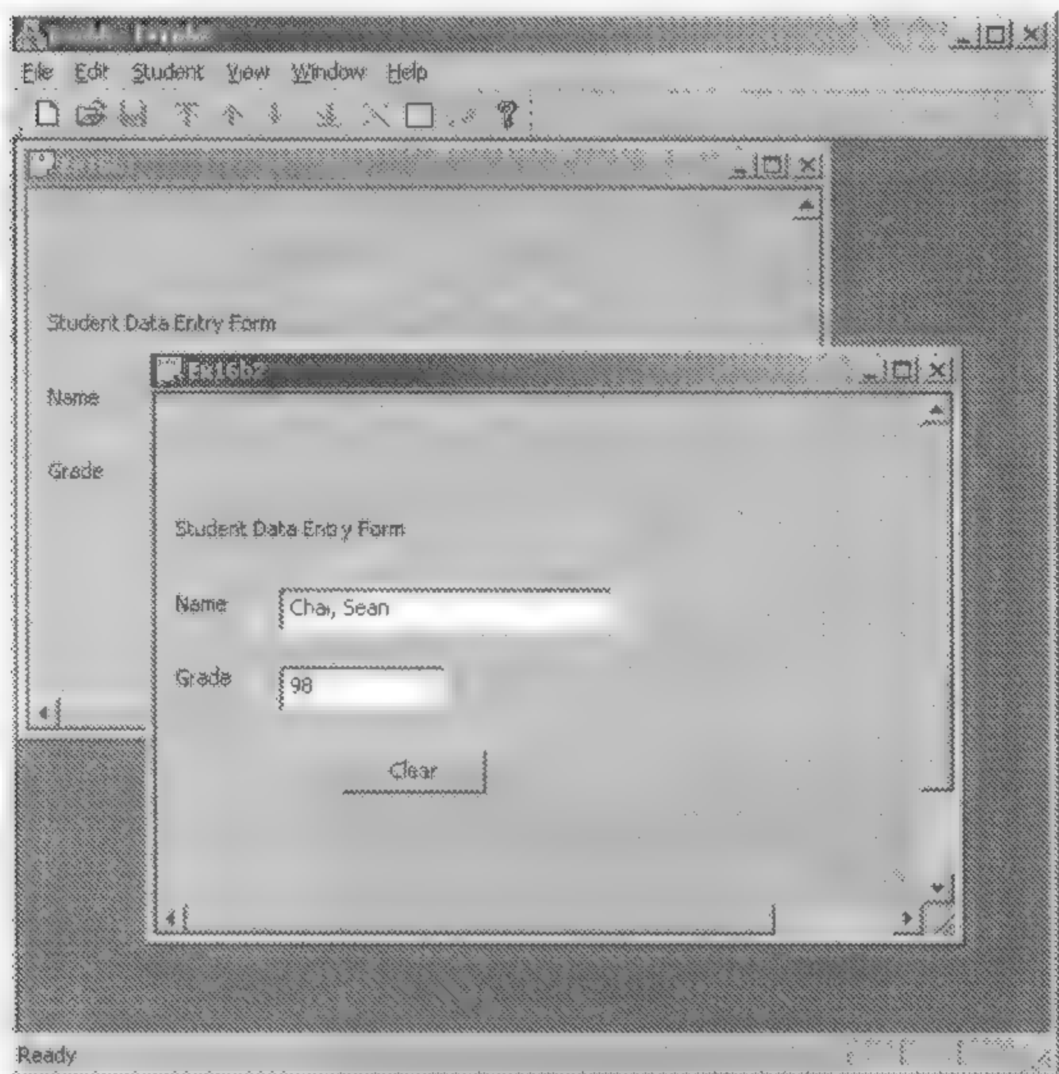


Рис. 16-4. *MDI-приложение Ex16b с двумя открытыми файлами*

Дочернее окно можно свернуть в значок внутри основного окна. Меню Window обеспечивает управление дочерними окнами при помощи команд:

Команды меню	Действие
New Window	Открывает дополнительное дочернее окно для текущего документа.
Cascade	Располагает существующие дочерние окна каскадом.
Tile	Располагает существующие дочерние окна так, чтобы они не перекрывались.
Arrange Icons	Упорядочивает значки окон в пределах окна-рамки.
<*имя документа*>	Активизирует соответствующее дочернее окно и помещает его поверх других окон.

Меню и панели инструментов в MDI-приложении динамичны. Когда все окна закрыты, состав меню File меняется, большинство кнопок на панели инструментов отключается, а в заголовке окна нет имени файла. Единственное, что можно сделать, — создать новый документ или загрузить существующий.

После запуска приложения создается новый документ с именем по умолчанию Ex16b1. Это имя формируется на основании значения параметра Doc Type Name (т. е. Ex16b), указанному на странице Document Template Strings в мастере MFC Application Wizard. Первому новому файлу присваивается имя Ex16b1, второму — Ex16b2 и т. д. Но при сохранении документа пользователь обычно задает более информативное имя.

MDI-приложения на базе MFC, как и многие коммерческие MDI-программы, при запуске автоматически создают новый пустой документ. (В этом смысле Visual C++ .NET — исключение.) Чтобы ваша программа при запуске открывала пустое окно-рамку, измените аргумент в вызове *ProcessShellCommand* в файле реализации класса приложения, как показано в примере Ex16b.

Объект «MDI-приложение»

Вас наверняка интересует, как работает MDI-программа и какой код заставляет ее вести себя не так, как SDI-приложение. Впрочем, процесс запуска приложений обоих типов во многом одинаков. Объект-приложение производного от *CWinApp* класса включает переопределенную функцию-член *InitInstance*. Она немного отличается от функции *InitInstance* SDI-приложения и начинается с вызова *AddDocTemplate*.

Класс шаблона MDI-документа

Вызов для создания шаблона MDI в функции *InitInstance* выглядит так:

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_EX16BTYPE,
    RUNTIME_CLASS(CEx16bDoc),
    RUNTIME_CLASS(CChildFrame), // специализированное дочернее окно-рамка MDI
    RUNTIME_CLASS(CEx16b));
AddDocTemplate(pDocTemplate);
```

В отличие от класса *CSingleDocTemplate*, который вы видели в Ex16a, *CMultiDocTemplate* позволяет программе использовать разные типы документов и допускает одновременное существование более чем одного объекта-документа. В этом суть MDI-программ.

Единственный вызов *AddDocTemplate*, показанный выше, позволяет MDI-программе поддерживать несколько дочерних окон, каждое из которых связано с отдельными объектами «документ» и «вид». Допускается наличие нескольких дочерних окон (и соответствующих объектов «вид»), связанных с одним объектом «документ». В этой главе мы поработаем только с одним классом «вид» и одним классом «документ». О применении нескольких классов «вид» и «документ» см. главу 18.

Примечание В процессе работы программы объект — шаблон документа поддерживает список активных объектов-документов, созданных с его помощью. «Проход» по этому списку обеспечивают функции-члены класса *CMultiDocTemplate* — *GetFirstDocPosition* и *GetNextDoc*. Для перехода от документа к его шаблону служит *CDocument::GetDocTemplate*.

Окно-рамка и дочернее окно в MDI-программе

В примерах SDI-программ присутствовал лишь один класс окна-рамки и один объект этого класса. Для SDI-приложений MFC Application Wizard генерирует класс *CMainFrame*, производный от класса *CFrameWnd*. В MDI-приложении два класса окна-рамки и множество объектов-рамок. Взаимосвязь окна-рамки и окна представления в MDI-приложении показана на рис. 16-5.

Базовый класс	Класс, генерируемый MFC Application Wizard	Число объектов	Меню и панели элементов управления	Содержит окно представления	Как создается объект
CMDIFrameWnd	*CMainFrame*	Только один	Да	Нет	В функции InitInstance класса приложения
CMDIChildWnd	*CChildFrame*	По одному на дочернее окно	Нет	Да	Каркасом приложений при открытии нового дочернего окна

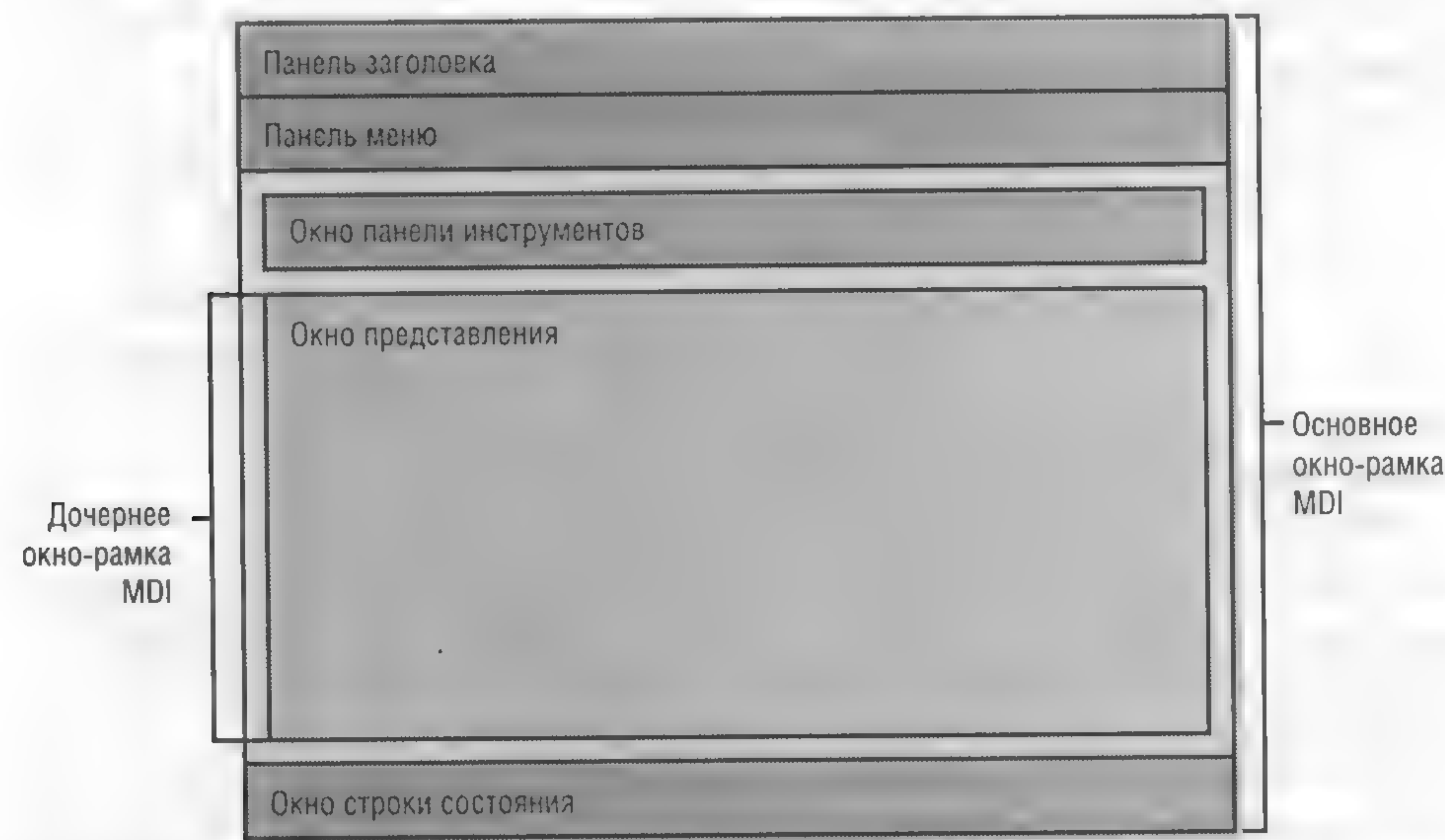


Рис. 16-5. *Взаимосвязь окна-рамки и окна представления в MDI-программе*

В SDI-приложении объект *CMainFrame* обрамляет приложение и содержит объект «вид». В MDI-приложении эти функции разделены. Теперь в *InitInstance* создается объект *CMainFrame*, а окно представления содержится внутри объекта *CChildFrame*. MFC Application Wizard генерирует такой код:

```
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
   return FALSE;
m_pMainWnd = pMainFrame;
:
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
```

Каркас приложений может создавать объекты *CChildFrame* динамически, так как указатель периода выполнения на класс *CChildFrame* передается конструктору *CMultiDocTemplate*.

Примечание Функция *InitInstance* MDI-приложения присваивает переменной-члену *m_pMainWnd* класса *CWinApp* указатель на основное окно-рамку. Это значит, что, если понадобится указатель на основное окно-рамку, вы сможете получить доступ к *m_pMainWnd* через глобальную функцию *AfxGetApp*[1].

Ресурсы основного окна-рамки и шаблона документа

В MDI-приложении (например, в Ex16b) два отдельных ресурса строк и меню, идентифицируемых константами *IDR_MAINFRAME* и *IDR_EX16BTYPE*. Первый ресурс используется, если основное окно-рамка пусто, а второй — если в нем есть дочерние окна. Вот как выглядят оба строковых ресурса, разбитые на подстроки:

```
IDR_MAINFRAME
    "Ex16b"                    // заголовок окна приложения

IDR_EX16BTYPE
    "\n"                       // (не используется)
    "Ex16b\n"                  // основа для имени документа по умолчанию
    "Ex16b\n"                  // имя типа документа
    "Ex16b Files (*.16b)\n"    // описание и фильтр для типа документа
    ".16b\n"                   // расширение для документов этого типа
    "Ex16b.Document\n"         // идентификатор типа файла в реестре
    "Ex16b.Document"           // описание типа файла в реестре
```

Примечание Компилятор ресурсов не поддерживает конкатенацию (сцепление) строк. Взгляните на содержимое файла Ex16b.rc: на самом деле подстроки объединены в одну длинную строку.

Заголовок окна приложения берется из строки *IDR_MAINFRAME*. При наличии открытого документа к заголовку добавляется имя файла этого документа. Последние две подстроки из *IDR_EX16BTYPE* поддерживают запуск с внедрением или операцией drag-and-drop.

Создание пустого документа

Функция *CWinApp::OnFileNew* позволяет создавать пустой документ. Как и в SDI-приложении, *InitInstance* в MDI-программе вызывает *OnFileNew* (через *ProcessShellCommand*). Однако на этот раз основное окно-рамка уже создано. Теперь же *OnFileNew*, вызывая функцию *OpenDocumentFile* класса *CMultiDocTemplate*, делает следующее.

1. Создает объект-документ, но не пытается загружать с диска его данные.
2. Создает объект (класса *CChildFrame*) дочернего окна-рамки MDI и формирует это окно, не отображая его на экране. Меню *IDR_MAINFRAME* в основном окне-рамке заменяется на меню *IDR_EX16BTYPE*. Идентификатор *IDR_EX16BTYPE*

---

[1] Для этого годится и глобальная функция *AfxGetMainWnd*. — *Прим. перев.*

определяет и ресурс значка, используемый при сворачивании дочернего окна в основном окне.

3. Создает объект «вид» и формирует окно представления, не выводя его на экран.
4. Устанавливает связь между *объектами* «документ», «дочерняя рамка» и «вид». Не путайте эти связи между объектами со связями между классами, которые устанавливает вызов *AddDocTemplate*.
5. Вызывает для объекта «документ» виртуальную функцию-член *OnNewDocument*.
6. Вызывает для объекта «вид» виртуальную функцию-член *OnInitialUpdate*.
7. Вызывает для объекта «дочерняя рамка» виртуальную функцию-член *ActivateFrame*, чтобы вывести на экран это окно-рамку и окно представления.

Функцию *OnFileNew* вызывает также команда New из меню File. В MDI-приложении функция *OnFileNew* работает так же, как и при вызове из *InitInstance*.

Примечание Некоторые из перечисленных выше функций вызываются не из *OpenDocumentFile*, а неявно — каркасом приложений.

Создание дополнительного окна представления для существующего документа

При выборе из меню Window команды New Window каркас приложений открывает новое дочернее окно, связанное с текущим документом. Соответствующая функция *OnWindowNew* из класса *CMDIFrameWnd* выполняет такие действия.

1. Создает объект дочернего окна-рамки (класса *CChildFrame*) и формирует само дочернее окно, не выводя его на экран.
2. Создает объект «вид» и формирует окно представления, не выводя его на экран.
3. Устанавливает связь между новым объектом «вид» и существующими объектами «документ» и «основное окно-рамка».
4. Вызывает для объекта «вид» виртуальную функцию-член *OnInitialUpdate*.
5. Вызывает для объекта дочернего окна-рамки виртуальную функцию-член *ActivateFrame*, чтобы вывести на экран это окно-рамку и окно представления.

Загрузка и сохранение документов

Документы в MDI-приложении загружаются и сохраняются практическим так же, как и в SDI-программе, но есть два важных отличия: всякий раз, когда документ загружается с диска, создается новый объект-документ, а при закрытии дочернего окна объект-документ уничтожается[1]. Об очистке содержимого документа перед загрузкой не беспокойтесь, но функцию *CDocument::DeleteContents* переопределите, чтобы класс документа можно было переносить в SDI-среду.

[1] Объект «документ» уничтожается, только если закрыто окно с его последним представлением. Если же есть окна с другими представлениями документа, он продолжает существовать. — *Прим. перев.*

Множественные шаблоны документов

MDI-приложение способно поддерживать множественные шаблоны документов при помощи нескольких вызовов *AddDocTemplate*. Каждый шаблон может задавать другую комбинацию классов «документ», «вид» и «дочерняя MDI-рамка». При вызове команды New из меню File каркас приложений выводит на экран список, позволяющий выбрать шаблон по имени, заданному в строковом ресурсе (подстрока типа документа). В SDI-приложении множественные вызовы *AddDocTemplate* не поддерживаются, так как объекты «документ», «вид» и «рамка» создаются только раз за все время жизни приложения.

Примечание При выполнении программы объект «приложение» ведет список объектов — активных шаблонов документов. «Проходить» по этому списку позволяют функции-члены *GetFirstDocTemplatePosition* и *GetNextDocTemplate* класса *CWinApp*. Вместе с функциями *GetFirstDocPosition* и *GetNextDoc* класса *CDocTemplate*, перечисляющими документы шаблона, они обеспечивают доступ ко всем объектам-документам в приложении.

Если список имен шаблонов вас не устраивает, отредактируйте меню File, добавив отдельные команды New для каждого типа документа. Закодируйте обработчики командных сообщений, как показано ниже, используя подстроку типа документа из каждого шаблона:

```
void CMyApp::OnFileNewStudent()
{
    OpenNewDocument("Studnt");
}
void CMyApp::OnFileNewTeacher()
{
    OpenNewDocument("Teachr");
}
```

Затем добавьте вспомогательную функцию *OpenNewDocument*:

```
BOOL CMyApp::OpenNewDocument(const CString& strTarget)
{
    CString strDocName;
    CDocTemplate* pSelectedTemplate;
    POSITION pos = GetFirstDocTemplatePosition();
    while (pos != NULL) {
        pSelectedTemplate = (CDocTemplate*) GetNextDocTemplate(pos);
        ASSERT(pSelectedTemplate != NULL);
        ASSERT(pSelectedTemplate->IsKindOf(
            RUNTIME_CLASS(CDocTemplate)));
        pSelectedTemplate->GetDocString(strDocName,
            CDocTemplate::docName);
        if (strDocName == strTarget) { // из строкового ресурса шаблона
            pSelectedTemplate->OpenDocumentFile(NULL);
            return TRUE;
        }
    }
    return FALSE;
}
```

Запуск MDI-программ из Windows Explorer и операцией drag-and-drop

Приложение запускается двойным щелчком значка документа MDI-приложения в Windows Explorer, если только оно уже не запущено, — иначе в уже работающем приложении открывается новое дочернее окно для выбранного документа. Для такого поведения надо вызвать *EnableShellOpen* в функции *InitInstance* класса приложения. Операция drag-and-drop работает примерно так же, как и в SDI-программах. Если перетащить файл из Explorer в основное окно-рамку MDI, программа откроет новое дочернее окно-рамку (с соответствующим документом и окном представления) — точно так же, как и по команде Open меню File. Как и в SDI-программах, расширение имен файлов указывается на странице Document Template Strings мастера MFC Application Wizard.

Пример Ex16b: MDI-приложение

Этот пример — MDI-версия программы Ex16a. В ней точно такой же код классов «документ» и «вид» и те же ресурсы (кроме имени программы). Однако код классов приложения и основного окна-рамки изменился. Ниже приведен лишь новый код, включая сгенерированный мастером MFC Application Wizard. Список файлов и классов в проекте Ex16b приведен в табл. 16-3.

Табл. 16-3. Файлы и классы программы Ex16b

Заголовочный файл	Файл исходного кода	Класс	Описание
Ex16b.h	Ex16b.cpp	*CEx16bApp*	Класс приложения (создан MFC Application Wizard)
		CAboutDlg	Диалоговое окно About
MainFrm.h	MainFrm.cpp	*CMainFrame*	Основное окно-рамка MDI-программы
ChildFrm.h	ChildFrm.cpp	*CChildFrame*	Дочернее окно-рамка MDI-программы
CEx16bDoc.h	CEx16bDoc.cpp	*CEx16bDoc*	Документ с данными о студенте (из Ex16a)
CEx16bView.h	Ex16bView.cpp	*CEx16bView*	Представление информации о студенте (из Ex16a)
Student.h	Student.cpp	*CStudent*	Запись о студенте (из Ex16a)
StdAfx.h	StdAfx.cpp		Предкомпилированные заголовочные файлы (с добавлением afxtempl.h)

CEx16bApp

В листинге исходного кода *CEx16bApp* функция-член *OpenDocumentFile* переопределена исключительно для вставки операторов *TRACE*. Кроме того, добавлено несколько строк перед вызовом *ProcessShellCommand* в *InitInstance*. В них проверяется аргумент для *ProcessShellCommand*, и при необходимости он изменяется, чтобы предотвратить автоматическое создание окна пустого документа при запуске программы.

Ex16b.h

```
[illegible]
```

Ex16b.cpp

```
[illegible]
```

```
[illegible]
```

см. след. стр.

```
[illegible]
    if(cmdInfo.m_nShellCommand == CCommandLineInfo::FileNew) {
        cmdInfo.m_nShellCommand = CCommandLineInfo::FileNothing;
    }
[illegible]
```

```
{
}
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()

// App command to run the dialog
void CEx16bApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

// CEx16bApp message handlers
CDocument* CEx16bApp::OpenDocumentFile(LPCTSTR lpszFileName)
{
    TRACE("CEx16bApp::OpenDocumentFile\n");
    return CWinApp::OpenDocumentFile(lpszFileName);
}
```

CMainFrame

Этот класс основного окна-рамки сходен с SDI-версией, однако его базовым классом является *CMDIFrameWnd*, а не *CFrameWnd*.

MainFrm.h

```
// MainFrm.h : interface of the CMainFrame class
//
#pragma once
class CMainFrame : public CMDIFrameWnd
{
    DECLARE_DYNAMIC(CMainFrame)
public:
    CMainFrame();
// Attributes
public:
// Operations
public:
// Overrides
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
    virtual ~CMainFrame();
```

см. след. стр.

```
[illegible]
```

MainFrm.cpp

```
[illegible]
```

```
{
   if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

   if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
      WS_CHILD | WS_VISIBLE | CBRS_TOP
      | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY
      | CBRS_SIZE_DYNAMIC) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
   {
      TRACE0("Failed to create toolbar\n");
      return -1;      // fail to create
   }
   if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
        sizeof(indicators)/sizeof(UINT)))
   {
      TRACE0("Failed to create status bar\n");
      return -1;      // fail to create
   }
   // TODO: Delete these three lines if you don't want the toolbar to
   // be dockable
   m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
   EnableDocking(CBRS_ALIGN_ANY);
   DockControlBar(&m_wndToolBar);
   return 0;
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   if( !CFrameWnd::PreCreateWindow(cs) )
      return FALSE;
   // TODO: Modify the Window class or styles here by modifying
   //  the CREATESTRUCT cs

   return TRUE;
}
// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
   CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
   CFrameWnd::Dump(dc);
}
#endif //_DEBUG
// CMainFrame message handlers
```

CChildFrame

Этот класс дочернего окна-рамки позволяет легко управлять характеристиками этого окна, изменяя код функции *PreCreateWindow*. Кроме того, вы можете создавать обработчики сообщений и переопределять другие виртуальные функции.

ChildFrm.h

```
// ChildFrm.h : interface of the CChildFrame class
//
#pragma once

class CChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CChildFrame)
public:
    CChildFrame();
// Attributes
public:
// Operations
public:
// Overrides
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
    virtual ~CChildFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
    DECLARE_MESSAGE_MAP()
public:
    virtual void ActivateFrame(int nCmdShow = -1);
};
```

ChildFrm.cpp

```
// ChildFrm.cpp : implementation of the CChildFrame class
//
#include "stdafx.h"
#include "[illegible]"
#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

```
// CChildFrame
IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)
BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
END_MESSAGE_MAP()

// [illegible]
CChildFrame::CChildFrame()
{
    // [illegible]
}
CChildFrame::~CChildFrame()
{
}
BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CMDIChildWnd::PreCreateWindow(cs) )
        return FALSE;
    return TRUE;
}
// CChildFrame diagnostics
#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}
void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}
#endif //_DEBUG

// [illegible]
void CChildFrame::ActivateFrame(int nCmdShow)
{
    TRACE("Entering CChildFrame::ActivateFrame\n");
    CMDIChildWnd::ActivateFrame(nCmdShow);
}
```

Тестирование приложения Ex16b

Соберите программу, запустите ее из Visual C++ .NET и создайте несколько документов. Попробуйте сохранить документы на диске, закрыть их и снова открыть. Проверьте также работу команды New Window из меню Window. Заметьте: теперь с одним документом связаны два окна представления и два дочерних окна-рамки. Завершите программу и откройте окно Windows Explorer. Рядом с именами созданных вами файлов должен отображаться значок документа. Дважды щелкните его и посмотрите, запустится ли Ex16b. Затем, запустив Explorer и Ex16b, перетащите документ из проводника в Ex16b. Открыла ли программа файл?

Работа с документами в MTI-приложениях

В Windows 2000 появился третий тип приложения: программа с множественными интерфейсами верхнего уровня (Multiple Top-Level Interface, MTI). Он применяется в Microsoft Office 2000 и Microsoft Office XP. MTI-приложения похожи на SDI-приложения, но каждая SDI-программа выполняется в отдельном окне, а в MTI один экземпляр приложения обслуживает все открытые окна. Когда пользователь создает новый файл, приложение открывает новое независимое окно верхнего уровня и соответствующие ему новый документ, но они закреплены за тем же экземпляром исполняемого приложения.

Пример Ex16c: MTI-приложение

Ex16c — это MTI-версия программы Ex16a. При создании этого примера в мастере MFC Application Wizard на странице Application Type нужно установить переключатель в положение Multiple Top-Level Documents, сбросить флажок Printing And Print Preview на странице Advanced Features и на странице Generated Classes выбрать в качестве базового класс *CFormView*.

В Ex16c использует тот же код классов документа и представления и те же ресурсы (кроме имен). Однако прикладной код и код класса основного окна-рамки другие. Код приложения Ex16c вы найдете на компакт-диске. Список файлов и классов в Ex16c примере показан в табл. 16-4.

Табл. 16-4. Файлы и классы примера Ex16c

Заголовочный файл	Файл с исходным кодом	Класс	Описание
Ex16c.h	Ex16c.cpp	*CEx16cApp*	Класс приложения (создан MFC Application Wizard)
		CAboutDlg	Диалоговое окно About
MainFrm.h	MainFrm.cpp	*CMainFrame*	Основное окно-рамка MTI-приложения
CEx16cDoc.h	CEx16cDoc.cpp	*CEx16cDoc*	Документ с данными о студенте (из Ex16a)
CEx16cView.h	Ex16cView.cpp	*CEx16cView*	Представление информации о студенте (из Ex16a)
Student.h	Student.cpp	*CStudent*	Запись о студенте (из Ex16a)
StdAfx.h	StdAfx.cpp		Предкомпилированные заголовочные файлы (с добавлением afxtempl.h)

В отличие от MDI- и SDI-приложений, MTI-приложение содержит в меню File команду New Frame. Она заставляет приложение открывать новое окно верхнего уровня. В листинге показана обработка команды New Frame.

```
void CEx16cApp::OnFileNewFrame()
{
    ASSERT(m_pDocTemplate != NULL);
    CDocument* pDoc = NULL;
    CFrameWnd* pFrame = NULL;

    // Создаем новый экземпляр документа, на который
```

```
    // ссылается переменная-член m_pDocTemplate.
    pDoc = m_pDocTemplate->CreateNewDocument();
    if (pDoc != NULL)
    {
        // В случае успеха создания создаем новое окно-рамку для документа
        pFrame = m_pDocTemplate->CreateNewFrame(pDoc, NULL);
        if (pFrame != NULL)
        {
            // Задаем заголовок и инициализируем документ.
            // В случае сбоя инициализации документа,
            // удаляем окно-рамку и документ.

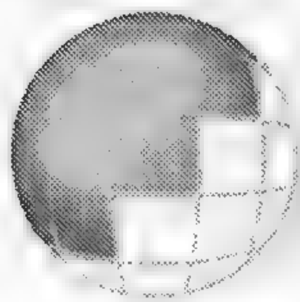
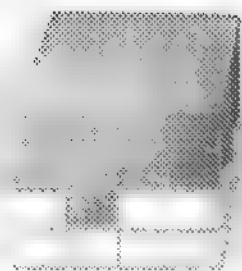
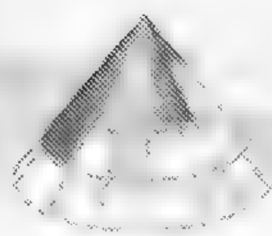
            m_pDocTemplate->SetDefaultTitle(pDoc);
            if (!pDoc->OnNewDocument())
            {
                pFrame->DestroyWindow();
                pFrame = NULL;
            }
            else
            {
                // В противном случае обновляем окно-рамку
                m_pDocTemplate->InitialUpdateFrame(pFrame, pDoc, TRUE);
            }
        }
    }

    // В случае неудачи удаляем документ и
    // выводим информационное окно для пользователя.
    if (pFrame == NULL || pDoc == NULL)
    {
        delete pDoc;
        AfxMessageBox(AFX_IDP_FAILED_TO_CREATE_DOC);
    }
}
```

Для управления документом, окном-рамкой и представлением в MTI-приложениях применяется класс *CMultiDocTemplate*. Заметьте: *OnFileNewFrame* самостоятельно создает новый документ и новое окно-рамку верхнего уровня, не полагаясь на каркас приложения для создания документа, окна-рамки и классов представления. В остальном MTI-приложения управляют своими документами и представлениями так же, как и SDI- и MDI-приложения.

Тестирование приложения Ex16c

Запустите приложение Ex16c и выберите команду New Frame в меню File. Заметьте, что новое окно-рамка находится рядом с существующим. Новое окно-рамка верхнего уровня содержит новый экземпляр документа, но документ связан с новым окном-рамкой (а не с новым дочерним окном-рамкой MDI, как в Ex16b).

ГЛАВА

17

Печать и предварительный просмотр

Если вы полагаетесь только на Win32 API, программирование печати станет для вас мукой. К счастью, каркас приложений библиотеки MFC существенно упрощает эту задачу. В нем также предусмотрена функция предварительного просмотра документов перед распечаткой, которая выполняет те же задачи, что и аналогичные функции в коммерческих Windows-программах, таких как Microsoft Word или Microsoft Excel.

В этой главе вы узнаете, как использовать MFC-функции печати и предварительного просмотра. Попутно вы получите представление о том, что происходит в Windows в процессе печати и чем этот процесс отличается от печати в MS-DOS. Сначала вы познакомитесь с программированием печати в режиме WYSIWYG, при котором принтерная распечатка практически идентична экранному изображению. Этот вариант требует аккуратного обращения с режимами преобразования координат в Windows. Потом мы объясним, как напечатать многостраничный отчет, выглядящий на бумаге совершенно не так, как на экране; кроме того, используя шаблон массива, вы структурируете свой документ так, чтобы программа смогла по требованию печатать любой заданный диапазон страниц.

Печать в Windows

Прежде программистам приходилось заботиться о настройке своих приложений для самых разных принтеров. Теперь об этом заботится Windows, в которой есть драйверы чуть ли не для всех принтеров. Кроме того, она обеспечивает единый пользовательский интерфейс для задач, связанных с печатью.

Стандартные диалоговые окна печати

Когда в Windows-приложении выбирают команду Print из меню File, на экране появляется стандартное диалоговое окно Print (рис. 17-1).

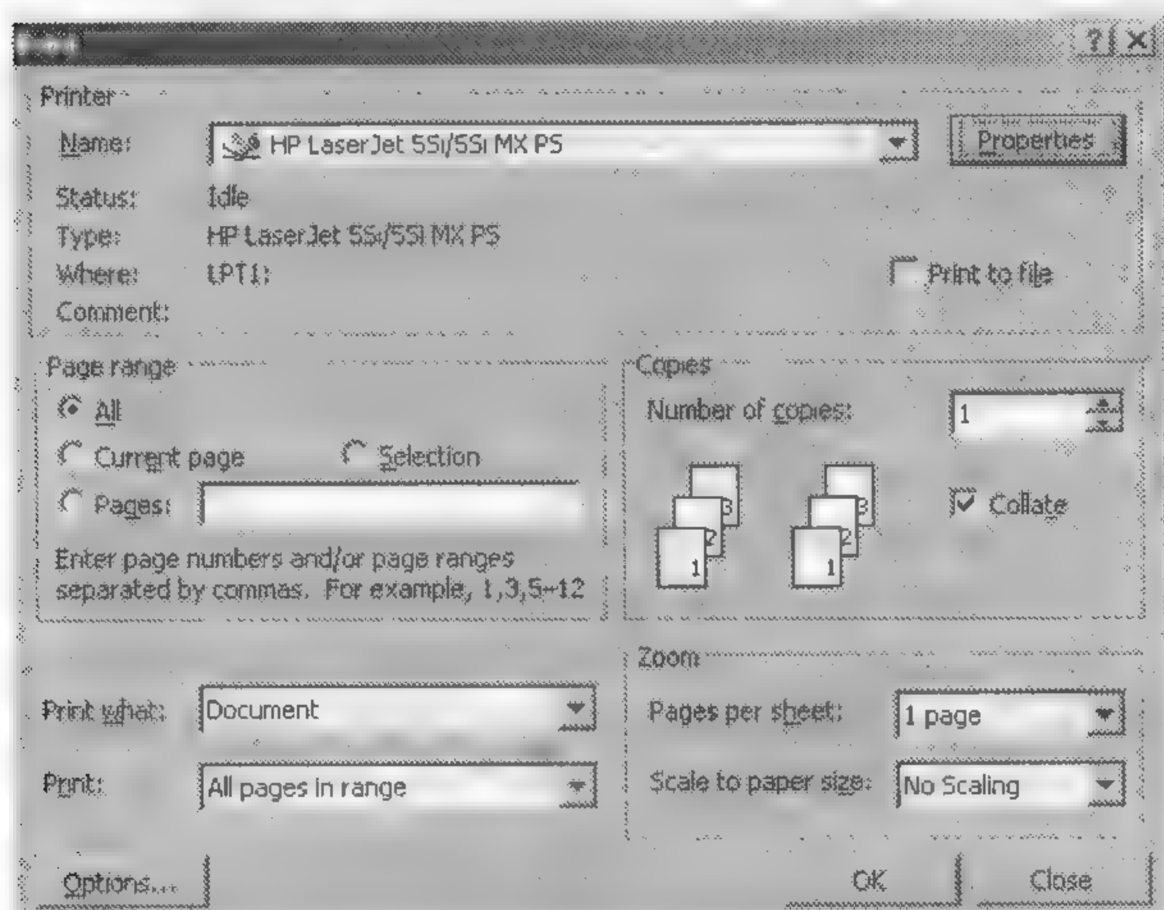


Рис. 17-1. *Стандартное диалоговое окно Print*

А если в диалоговом окне Print щелкнуть кнопку Properties, откроется окно Document Properties (рис. 17-2).

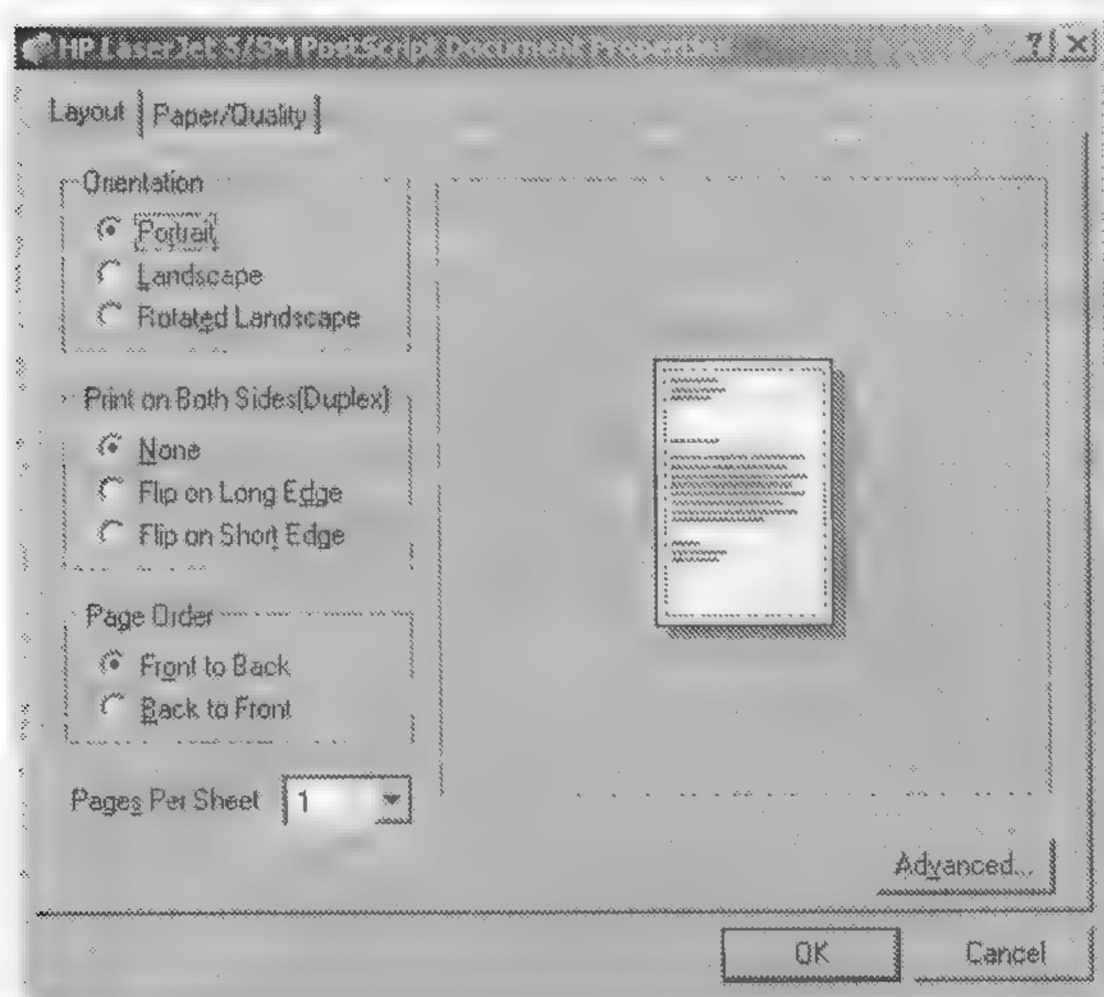


Рис. 17-2. *Диалоговое окно Document Properties*

В процессе печати программа выводит стандартное диалоговое окно с информацией о состоянии принтера[1].

[1] Не совсем так — это окно отображается при пересылке данных в службу печати (спулер), а не на весь период (обычно более длительный) собственно печати файла на принтере. — *Прим. перев.*

Интерактивный выбор страниц для печати

Если вы занимались обработкой данных, то, наверное, привыкли к пакетному режиму печати. Программа считывает запись, форматирует ее и печатает выбранную информацию как строку в отчете. После распечатки, скажем, 50 строк, программа заставляет принтер вытолкнуть лист бумаги и начать печатать новую страницу. В таких случаях обычно полагают, что отчет печатается целиком, и не предусматривают возможности выбора страниц в интерактивном режиме.

Но при печати в Windows разбиение на страницы играет большую роль (рис. 17-1). Программа должна реагировать на выбор страниц пользователем, вычисляя, какую именно информацию напечатать, поэтому вы должны соответственно структурировать данные в своем приложении.

Помните список студентов из главы 16? Что, если в нем окажется 1000 студентов и пользователю понадобится 5-я страница отчета? Допустим, запись о студенте укладывается в одну строку, а на одной странице умещается 50 строк. Тогда на пятой странице должны быть строки с 201 по 250. Если вы используете MFC-класс списка, вам придется, прежде чем приступить к печати, «пролистать» первые 200 элементов списка. Понятно, что в данной ситуации список — далеко не идеальная структура. А если вместо него применить массив? Класс *CObArray* (или один из классов-шаблонов массива) позволит обратиться прямо к 201-й записи о студенте.

Отнюдь не в каждом приложении переменные-члены жестко связаны с определенным числом печатных строк. Представьте, что в записи о студенте есть поле «биография» в несколько печатных строк. Так как заранее неизвестно, сколько строк занимает биография конкретного студента, придется просматривать весь файл, чтобы определить границы страниц. Но ваша программа станет работать гораздо эффективнее, если сможет «запоминать» эти границы по мере их вычисления.

Экранные и печатные страницы

Часто желательно, чтобы отпечатанная страница соответствовала изображению на экране. Гарантировать этого нельзя. Однако шрифты TrueType позволяют добиться близкого соответствия. Если вы работаете с полноразмерными листами бумаги, окно должно быть больше размера экрана. Поэтому для просмотра печатаемых изображений идеально подходит класс *CScrollView*.

Впрочем, иногда об экранных страницах можно не заботиться. Скажем, ваш класс «вид» хранит данные в каком-то окне списка, или их вообще не надо выводить на экран. В таких случаях в программу обычно закладывают логику «автономной» печати, при которой данные просто извлекаются из документа и отсылаются на принтер. Конечно, программа должна корректно обрабатывать запрос пользователя на печать не всего документа, а лишь какого-то диапазона страниц. Запросив у драйвера принтера размер бумаги и ориентацию листов (книжная или альбомная), вы сможете корректировать разбиение документа на страницы.

Предварительный просмотр перед печатью

Предварительный просмотр перед печатью (print preview), поддерживаемый MFC-библиотекой, позволяет увидеть на экране границы страниц и концы строк *точ-*

но в тех местах, где они получатся при распечатке документа на выбранном принтере. Шрифт, особенно мелкий, может выглядеть несколько странно, но это не проблема.

Предварительный просмотр перед печатью поддерживает библиотека MFC, а не Windows. Реализация поддержки потребовала от разработчиков библиотеки колоссальных усилий. Эта функция анализирует каждый символ, определяя его позицию на основе контекста принтера. Подобрав более-менее подходящий шрифт, она выводит символ в окно предварительного просмотра.

Программирование вывода на печать

Каркас приложений берет на себя большую часть работы по поддержке печати и предварительного просмотра. Чтобы эффективно использовать принтер, надо знать порядок вызовов функций и понимать, какие из них и когда переопределять.

Контекст принтера и функция *CView::OnDraw*

При печати на принтере программа использует объект «контекст устройства» класса *CDC*. Не беспокойтесь о том, откуда возьмется этот объект, — его создает каркас приложений, передавая затем как параметр в функцию *OnDraw* вашего класса «вид». Если программа просто копирует содержимое экрана на принтер, *OnDraw* может выполнять две задачи. При выводе на экран *OnPaint* вызывает *OnDraw*, передавая ей контекст дисплея, а при печати *OnPrint* другая виртуальная функция класса *CView* вызывает *OnDraw*, передавая ей контекст принтера. Один вызов функции *OnPrint* позволяет отпечатать одну страницу целиком.

В режиме предварительного просмотра параметр функции *OnDraw* служит указателем на объект «контекст устройства» класса *CPreviewDC*. Функции *OnPrint* и *OnDraw* работают одинаково независимо от того, печатаете вы или просматриваете документ перед печатью.

Функция *CView::OnPrint*

Вы уже убедились, что функция *OnPrint* (базового класса) вызывает *OnDraw* и что та способна использовать контекст как дисплея, так и принтера. Перед вызовом *OnPrint* нужно установить режим преобразования координат. Чтобы печатать элементы, которые не обязательно показывать на экране (скажем, титульную страницу, верхние и нижние колонтитулы), функцию *OnPrint* можно переопределить. Она получает два указателя: на контекст устройства и на структуру *CPrintInfo*, где хранятся размеры страницы, номер текущей страницы и максимальный номер страницы.

Переопределяя функцию *OnPrint*, можно вообще не вызывать *OnDraw*, когда логика печати полностью независима от логики вывода на экран. Для каждой печатаемой страницы каркас приложений вызывает *OnPrint* по одному разу — с указанием ее номера в структуре *CPrintInfo*.

Подготовка контекста устройства: функция *CView::OnPrepareDC*

Если вам нужен режим преобразования координат на экране, отличный от *MM_TEXT* (а так оно обычно и бывает), лучше установить его в функции *OnPrepareDC* класса «вид». Вы сами переопределяете эту функцию, если ваш класс «вид» наследует напрямую классу *CView*, но, если он является производным *CScrollView*, функция уже переопределена. *OnPaint* вызывает *OnPrepareDC* прямо перед вызовом *OnDraw*. Если же вы печатаете на принтере, обращение к *OnPrepareDC* выполняется перед тем, как каркас приложений вызывает *OnPrint*. Так что программа устанавливает режим преобразования координат и перед прорисовкой экранного изображения, и перед печатью страницы.

Второй параметр функции *OnPrepareDC* — указатель на структуру *CPrintInfo*. Он действителен, только если *OnPrepareDC* вызывается до печати. Проверяет его достоверность функция-член *CDC::IsPrinting*. Она особенно удобна, если через *OnPrepareDC* вы устанавливаете на экране и на принтере разные режимы преобразования координат.

Если вы заранее не знаете, сколько страниц придется печатать, переопределите функцию *OnPrepareDC* так, чтобы она отыскивала конец документа и сбрасывала флажок *m_bContinuePrinting* в структуре *CPrintInfo*. Если этот флажок равен *FALSE*, функция *OnPrint* не вызывается, а управление передается в конец цикла печати.

Начало и конец печати

Когда начинается процесс печати, каркас приложений вызывает две функции класса *CView* — *OnPreparePrinting* и *OnBeginPrinting*. Первая вызывается перед открытием диалогового окна Print. (При установленном флажке Printing and Print Preview мастер MFC Application Wizard генерирует функции *OnPreparePrinting*, *OnBeginPrinting* и *OnEndPrinting*.) Функция *OnPreparePrinting* вызывается перед отображением окна Print. Если вам известно минимальное и максимальное число страниц, вызовите из *OnPreparePrinting* функции *CPrintInfo::SetMinPage* и *CPrintInfo::SetMaxPage*. Числа, которые вы передаете этим функциям, появятся в диалоговом окне Print, и пользователь сможет изменить их.

Функция *OnBeginPrinting* вызывается после закрытия диалогового окна Print. Она переопределяется для создания GDI-объектов, например, необходимых для печати шрифтов. Программа будет работать быстрее, если создать шрифт заранее, а не повторять эту операцию для каждой страницы.

Функция *CView::OnEndPrinting* вызывается в конце печати — после того, как отпечатана последняя страница. Ее переопределяют для уничтожения GDI-объектов, созданных в *OnBeginPrinting*.

В табл. 17-1 показаны важнейшие для цикла печати функции класса *CView*, которые обычно переопределяют.

Табл. 17-1. Переопределяемые функции цикла печати в классе *CView*

Функция	Для чего обычно переопределяется
OnPreparePrinting	Установка минимального и максимального числа страниц
OnBeginPrinting	Создание GDI-объектов
OnPrepareDC (для каждой страницы)	Установка режима преобразования координат и определение момента конца печати
OnPrint (для каждой страницы)	Выполнение подготовительных операций, связанных с выводом на печать, и вызов *OnDraw*
OnEndPrinting	Удаление GDI-объектов

Пример Ex17a: печать в режиме WYSIWYG

Эта программа отображает на экране и печатает одну страницу текста. Отпечатанная страница совпадает с изображением на экране. И для принтера, и для дисплея задается режим преобразования координат *MM_TWIPS*. В исходном виде программа выводит текст в *фиксированную область печати* (fixed printable area rectangle), но потом мы модернизируем ее так, чтобы она подстраивалась к области печати, поддерживаемой драйвером принтера.

1. **В окне MFC Application Wizard создайте Ex17a.** Примите параметры по умолчанию, но на странице Generated Classes переименуйте классы «вид» *CStringView* и «документ» *CPoemDoc*. В качестве базового для *CStringView* выберите класс *CScrollView*. Обратите внимание, что создается MDI-приложение.
2. **Добавьте переменную-член типа *CStringArray* в класс *CPoemDoc*.** Отредактируйте заголовочный файл PoemDoc.h, добавив строку:

```
public:
    CStringArray m_stringArray;
```

   Данные документа хранятся в массиве строк. MFC-класс *CStringArray* содержит массив объектов класса *CString*, доступных по индексу (нумерация начинается с 0). Максимальную размерность массива при объявлении указывать не надо, потому что он динамический.
3. **Добавьте переменную-член типа *CRect* в класс *CStringView*.** Отредактируйте заголовочный файл StringView.h, добавив строку:

```
private:
    CRect m_rectPrint;
```

4. **Отредактируйте три функции-члена класса *CPoemDoc* в файле PoemDoc.cpp.** MFC Application Wizard создал заготовки функций *OnNewDocument* и *Serialize*, но нам придется в окне Properties утилиты Class View переопределить также функцию *DeleteContents*. Инициализацию документа-«стихотворения» мы предусмотрим в переопределенной функции *OnNewDocument*. *DeleteContents* вызывается из *CDocument::OnNewDocument*, так что благодаря вызову сначала функции базового класса стихотворение не пропадет. [Кстати, это фрагмент 20-го стихотворения из книги Лоренса Ферлингетти (Lawrence Ferlinghetti) «A Coney Island of the Mind».] Если вам не нравится текст — возьмите

другое произведение или описание любимой Win32-функции. Добавьте выделенный код:

```
BOOL CPoemDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_stringArray.SetSize(10);
    m_stringArray[0] = "The pennycandystore beyond the El";
    m_stringArray[1] = "is where I first";
    m_stringArray[2] = "                    fell in love";
    m_stringArray[3] = "                        with unreality";
    m_stringArray[4] = "Jellybeans glowed in the semi-gloom";
    m_stringArray[5] = "of that september afternoon";
    m_stringArray[6] = "A cat upon the counter moved among";
    m_stringArray[7] = "                       the licorice sticks";
    m_stringArray[8] = "                    and tootsie rolls";
    m_stringArray[9] = "             and Oh Boy Gum";

    return TRUE;
}
```

Примечание Класс *CStringArray* поддерживает динамические массивы, но здесь мы используем объект *m_stringArray* так, будто это статический массив с 10 элементами.

При закрытии документа каркас приложений вызывает виртуальную функцию *DeleteContents* класса документа; при этом строки в массиве удаляются. *CStringArray* содержит собственно объекты, а *CObArray* — указатели на них. Важность этого различия станет ясна, когда придет пора удалять элементы массива. Функция *RemoveAll* уничтожает строковые объекты так:

```
void CPoemDoc::DeleteContents()
{
    // вызывается перед вызовом OnNewDocument и при закрытии документа
    m_stringArray.RemoveAll();
}
```

Сериализация в этом примере не существенна, но приведенная ниже функция иллюстрирует, насколько проста эта операция над строками. Каркас приложений вызывает *DeleteContents* перед загрузкой из архива, так что вам нет нужды беспокоиться об очистке массива. Введите выделенный код:

```
void CPoemDoc::Serialize(CArchive& ar)
{
    m_stringArray.Serialize(ar);
}
```

5. **Отредактируйте функцию *OnInitialUpdate* в StringView.cpp.** Эту функцию следует переопределить для всех классов, производных от *CScrollView*. Она уста-

навливает логический размер окна и режим преобразования координат. Добавьте выделенный код:

```
void CStringView::OnInitialUpdate()
{
   CScrollView::OnInitialUpdate();
   CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
   CSize sizePage(sizeTotal.cx / 2,
              sizeTotal.cy / 2);   // прокрутка страницы
   CSize sizeLine(sizeTotal.cx / 100,
              sizeTotal.cy / 100); // прокрутка строки
   SetScrollSizes(MM_TWIPS, sizeTotal, sizePage, sizeLine);
}
```

6. **Отредактируйте функцию *OnDraw* в StringView.cpp.** Функция *OnDraw* класса *CStringView* выводит изображение и на экран, и на принтер. Она не только показывает строки стихотворения шрифтом Times New Roman (размером 10 пт), но и очерчивает область печати и формирует какое-то подобие линеек по верхнему и левому полям. Функция предполагает применение режима *MM_TWIPS*, при котором 1 дюйм равен 1440 единицам. Добавьте выделенный код:

```
void CStringView::OnDraw(CDC* pDC)
{
   int    i, j, nHeight;
   CString   str;
   CFont   font;
   TEXTMETRIC tm;

   CPoemDoc* pDoc = GetDocument();
   // Рисуем рамку - чуть меньше, чтобы избежать отсечения
   pDC->Rectangle(m_rectPrint + CRect(0, 0, -20, 20));
   // Рисуем вертикальную и горизонтальную линейки
   j = m_rectPrint.Width() / 1440;
   for (i = 0; i <= j; i++) {
      str.Format("%02d", i);
      pDC->TextOut(i * 1440, 0, str);
   }
   j = -(m_rectPrint.Height() / 1440);
   for (i = 0; i <= j; i++) {
      str.Format("%02d", i);
      pDC->TextOut(0, -i * 1440, str);
   }
   // напечатать текст на полдюйма ниже и правее...
   // шрифтом Times New Roman, 10 пунктов
   font.CreateFont(-200, 0, 0, 0, 400, FALSE,
               FALSE, 0, ANSI_CHARSET,
               OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
               DEFAULT_QUALITY, DEFAULT_PITCH | FF_ROMAN,
               "Times New Roman");
   CFont* pOldFont = (CFont*) pDC->SelectObject(&font);
   pDC->GetTextMetrics(&tm);
```

```
    nHeight = tm.tmHeight + tm.tmExternalLeading;
    TRACE("font height = %d, internal leading = %d\n",
        nHeight, tm.tmInternalLeading);
    j = pDoc->m_stringArray.GetSize();
    for (i = 0; i < j; i++) {
        pDC->TextOut(720, -i * nHeight - 720,
                pDoc->m_stringArray[i]);
    }
    pDC->SelectObject(pOldFont);
    TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n",
        pDC->GetDeviceCaps(LOGPIXELSX),
        pDC->GetDeviceCaps(LOGPIXELSY));
    TRACE("HORZSIZE = %d, VERTSIZE = %d\n",
        pDC->GetDeviceCaps(HORZSIZE),
        pDC->GetDeviceCaps(VERTSIZE));
}
```

7. **Отредактируйте функцию *OnPreparePrinting* в StringView.cpp.** Она устанавливает максимальное число печатаемых страниц. В нашем примере печатается только одна страница. В переопределенной функции *OnPreparePrinting* надо обязательно вызвать функцию *DoPreparePrinting* базового класса. Введите выделенный код:

```
BOOL CStringView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(1);
    return DoPreparePrinting(pInfo);
}
```

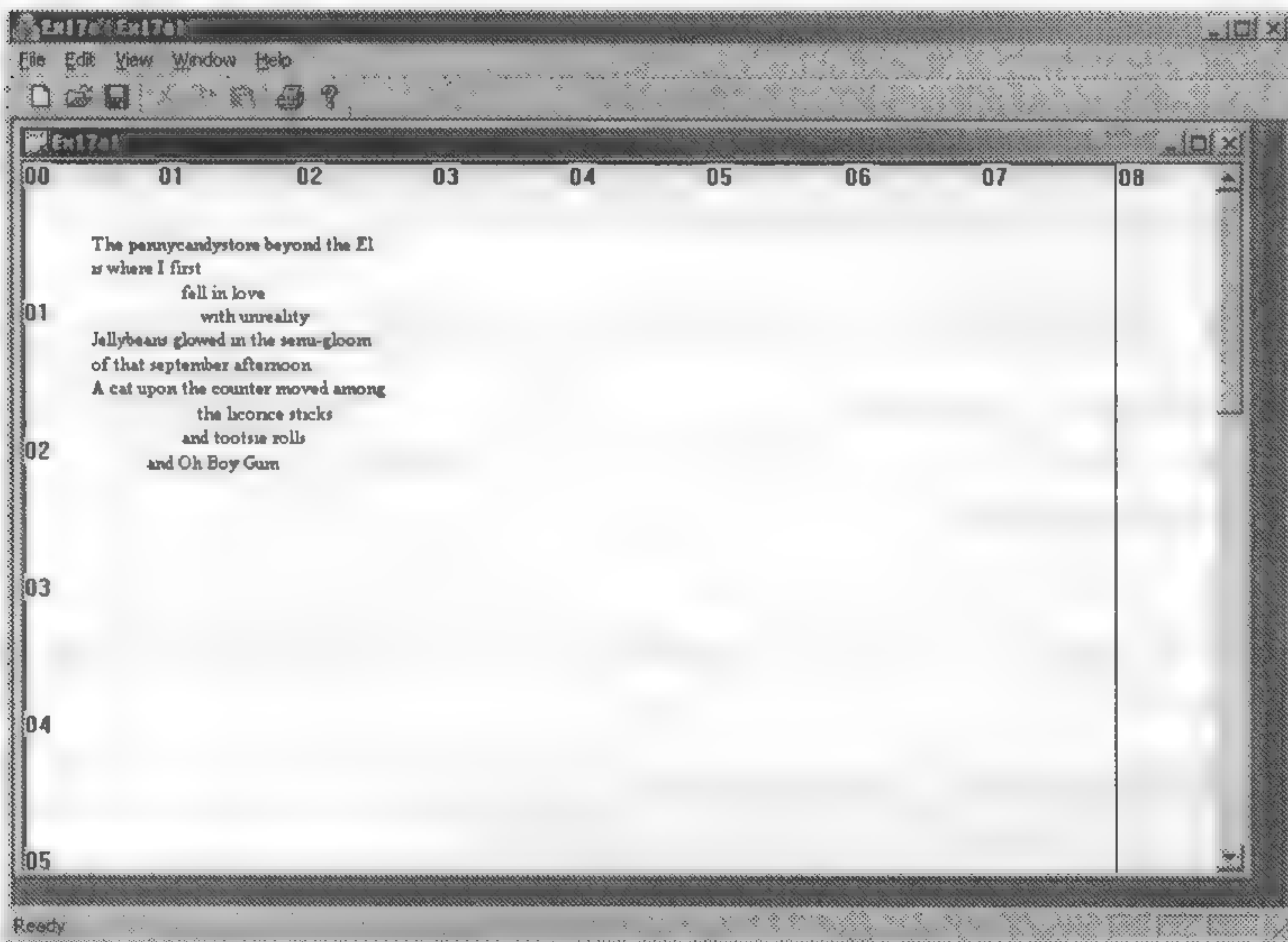
8. **Отредактируйте конструктор в StringView.cpp.** Начальная область печати должна составлять 8×15 дюймов и выражаться в твипах (1 дюйм = 1440 твипов). Добавьте выделенный код:

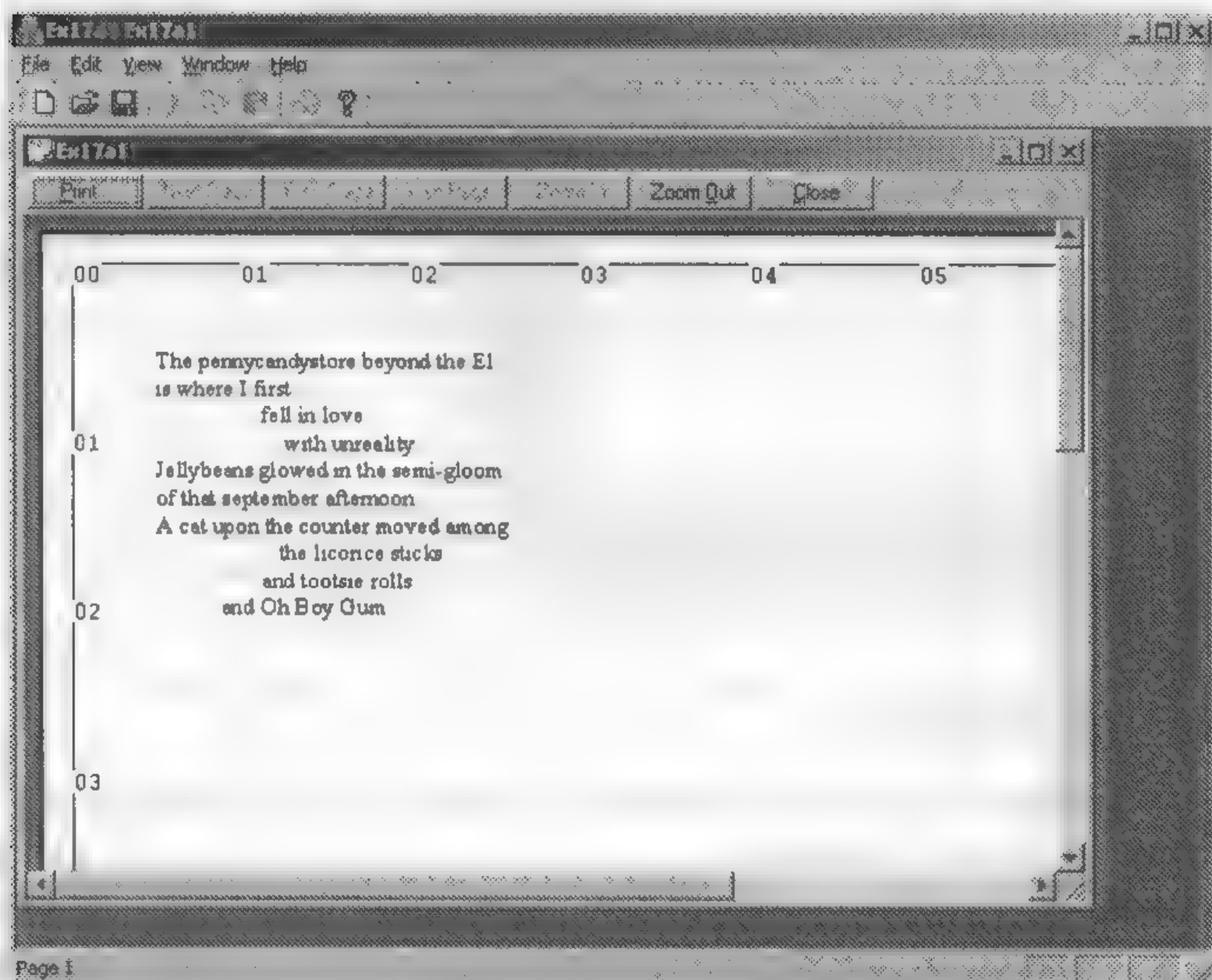
```
CStringView::CStringView() : m_rectPrint(0, 0, 11520, -21600){
}
```

9. **Соберите и протестируйте приложение.** Если запустить приложение Ex17a под Windows NT/2000/XP с низким экранным разрешением, дочернее MDI-окно должно выглядеть, как рисунке. (При более высоком разрешении или под Windows 95/98 текст кажется более крупным.)

   Текст мелковат, правда? Пойдем дальше: выберем из меню File пункт Print Preview и увеличим изображение, дважды щелкнув «лупу». Результат показан на следующей странице.

   Помните «логические твипы» из главы 6? Сейчас мы попробуем с их помощью увеличить изображение на дисплее, не меняя его размер при печати. Это требует дополнительных усилий, так как класс *CScrollView* не рассчитан на нестандартные режимы преобразования координат. Придется заменить базовый класс представления: вместо *CScrollView* взять *CLogScrollView*, который один из авторов создал, изменив исходный MFC-код в ViewScrl.cpp. Файлы LogScrollView.h и LogScrollView.cpp находятся в каталоге \vcppnet\ Ex17a на компакт-диске.





10. **Вставьте класс *CLogScrollView* в проект.** Скопируйте файлы LogScrollView.h и LogScrollView.cpp с компакт-диска (если не сделали этого раньше). В меню Project выберите Add Existing Item, а затем в открывшемся окне выберите скопированные файлы и щелкните OK, чтобы внести их в проект.

11. **Отредактируйте заголовочный файл StringView.h.** Добавьте в начало файла:

```
#include "LogScrollView.h"
```

Затем замените строку:

```
class CStringView : public CScrollView
```

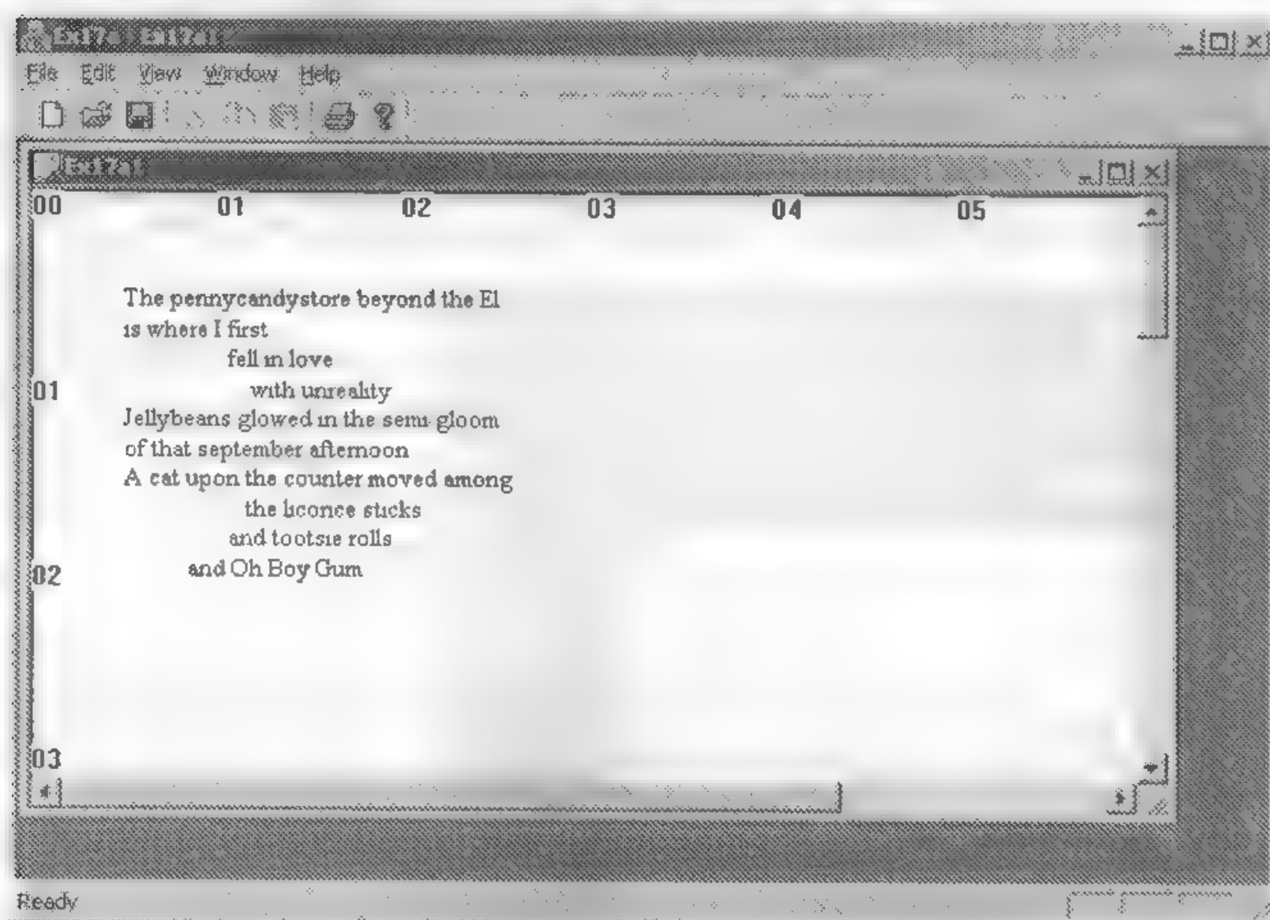
на:

```
class CStringView : public CLogScrollView
```

12. **Отредактируйте файл StringView.cpp.** Повсеместно замените все вхождения *CScrollView* на *CLogScrollView*. Затем отредактируйте функцию *OnInitialUpdate*. Отредактированный код намного короче:

```
void CStringView::OnInitialUpdate()
{
    CLogScrollView::OnInitialUpdate();
    CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
    SetLogScrollSizes(sizeTotal);
}
```

13. **Снова соберите и протестируйте приложение.** Теперь на экране должно быть нечто вроде:



Определение области печати

Ex17a печатает в фиксированной области, соответствующей настройке лазерного принтера для печати в книжной ориентации на листах размером 8,5×11 дюймов (формат Letter). А если вы загрузили бумагу формата A4 или выбрали альбомную ориентацию? Программа должна уметь подстраиваться под новые параметры.

Определить область печати принтера сравнительно несложно. Помните указатель на структуру *CPrintInfo*, передаваемый в *OnPrint*? В этой структуре есть поле *m_rectDraw*, в котором хранятся логические координаты области печати. К ней и обращается функция *OnDraw*. Правда, определить эту область, пока не начнется печать, нельзя, поэтому надо установить для *OnDraw* какую-то область по умолчанию, чтобы использовать ее до начала печати.

Если вы хотите, чтобы Ex17a определяла область печати принтера, переопределите *OnPrint* в окне Properties утилиты Class View, а затем напишите ее код:

```
void CStringView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    m_rectPrint = pInfo->m_rectDraw;
```

```
    SetLogScrollSizes(CSize(m_rectPrint.Width(),
                      -m_rectPrint.Height()));
    CLogScrollView::OnPrint(pDC, pInfo);
}
```

Еще раз о классах-шаблонах наборов: класс *CArray*

В программе Ex15b (см. главу 15) мы работали с *CTypedPtrList* — классом-шаблоном набора из MFC-библиотеки, в котором мы хранили список указателей на объекты *CStudent*. Другой такой класс, *CArray*, пригодится для примера Ex17b. Он отличается от *CTypedPtrList* по двум позициям. Во-первых, как и *CStringArray* в Ex17a, это массив с элементами, доступными по индексу. Во-вторых, он хранит не указатели на объекты, а сами объекты. В программе Ex17b элементы массива — это объекты класса *CRect*. Класс элемента не должен быть производным от *CObject*, а в случае с *CRect* это как раз так.

Как и в Ex15b, оператор *typedef* упрощает работу с шаблоном. Мы напишем:

```
typedef CArray<CRect, CRect&> CRectArray;
```

и определим тем самым класс массива, содержащий объекты *CRect*, ссылки на которые будут принимать функции этого класса. (Передавать 32-разрядные указатели эффективнее, чем копировать 128-разрядные объекты.) Чтобы использовать шаблон массива, надо объявить экземпляр класса *CRectArray*, а затем вызвать функции-члены класса *CArray*, например *SetSize*. Кроме того, для доступа к элементам этого массива можно применить оператор [] класса *CArray*.

Классы-шаблоны *CArray*, *CList* и *CMap* просты в обращении, если прост класс элементов. Класс *CRect* удовлетворяет этому условию, так как не содержит указателей. Для сериализации всех элементов в наборе каждый класс-шаблон вызывает глобальную функцию *SerializeElements*, предлагаемую по умолчанию, которая осуществляет *побитовое* копирование объекта в архив и из архива.

Если ваш класс элементов содержит указатели или что-то еще, усложняющее его, придется написать свою функцию *SerializeElements*. Скажем, для массива прямоугольников эта функция выглядела бы так (на самом деле это не нужно):

```
void AFXAPI SerializeElements(CArchive& ar, CRect* pNewRects, int nCount)
{
    for (int i = 0; i < nCount: i++, pNewRects++) {
        if (ar.IsStoring()) {
            ar << *pNewRects;
        }
        else {
            ar >> *pNewRects;
        }
    }
}
```

Обнаружив эту функцию, компилятор использует ее вместо *SerializeElements*, определенной в шаблоне, но только если прототип функции *SerializeElements* указан до объявления класса шаблона.

Примечание Классы-шаблоны зависят и от двух других глобальных функций: *ConstructElements* и *DestructElements*. С Visual C++ 4.0 эти функции вызывают для каждого объекта конструктор и деструктор класса элементов, так что заменять их не надо.

Пример Ex17b: программа печати многих страниц

В этом примере документ содержит массив из 50 объектов *CRect*, описывающих окружности, которые случайным образом разбросаны в прямоугольной области 6×6 дюймов и имеют произвольные диаметры (не менее полдюйма) — получается нечто напоминающее мыльные пузыри. В то же время программа выводит на принтер не сами окружности, а координаты соответствующих им объектов *CRect* в числовой форме — по 12 на каждой странице с колонтитулами.

1. **Средствами MFC Application Wizard создайте проект Ex17b.** Примите параметры по умолчанию, но на странице Application Type мастера установите переключатель в положение Single document.
2. **Отредактируйте заголовочный файл StdAfx.h.** Добавьте объявление классов-шаблонов наборов MFC. Для этого вставьте строку:

```
#include <afxtempl.h>
```

3. **Отредактируйте заголовочный файл Ex17bDoc.h.** В примере Ex17a данные документа состояли из строк, хранившихся в наборе *CStringArray*. Поскольку мы используем шаблон набора для прямоугольников, ограничивающих эллипсы, нам нужен оператор *typedef* (вне объявления класса):

```
typedef CArray<CRect, CRect&> CRectArray;
```

Теперь объявите в Ex17bDoc.h открытые переменные-члены:

```
public:
    enum { nLinesPerPage = 12 };
    enum { nMaxEllipses = 50 };
    CRectArray m_ellipseArray;
```

Операторы перечисления — «объектно-ориентированные» заменители операторов *#define*.

4. **Отредактируйте файл реализации Ex17bDoc.cpp.** Переопределенная функция *OnNewDocument* инициализирует массив эллипсов произвольными значениями, а функция *Serialize* считывает и записывает весь массив. Заготовку этих функций генерирует MFC Application Wizard. Функция *DeleteContents* не нужна, потому что оператор [] класса *CArray* записывает новый объект *CRect* вместо существующего. Добавьте выделенный код:

```
BOOL CEx17bDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    int n1, n2, n3;
```

```
    // Создаем 50 произвольных окружностей
    srand((unsigned) time(NULL));
    m_ellipseArray.SetSize(nMaxEllipses);

    for (int i = 0; i < nMaxEllipses; i++) {
        n1 = rand() * 600 / RAND_MAX;
        n2 = rand() * 600 / RAND_MAX;
        n3 = rand() * 50  / RAND_MAX;
        m_ellipseArray[i] = CRect(n1, -n2, n1 + n3, -(n2 + n3));
    }

    return TRUE;
}

void CEx17bDoc::Serialize(CArchive& ar)
{
    m_ellipseArray.Serialize(ar);
}
```

5. **Измените заголовочный файл Ex17bView.h.** Используя доступные в окне Class View мастера Add Member Variable Wizard и Add Member Function Wizard добавьте переменную-член и два прототипа функций, приведенные ниже. Add Member Function Wizard одновременно сгенерирует в Ex17bView.cpp шаблоны этих функций.

```
public:
    int m_nPage;
private:
    void PrintPageHeader(CDC* pDC);
    void PrintPageFooter(CDC* pDC);
```

   Переменная-член *m_nPage* хранит номер текущей страницы документа. Закрытые функции предназначены для подпрограмм, формирующих верхние и нижние колонтитулы.

6. **Отредактируйте функцию *OnDraw* в Ex17bView.cpp.** Переопределенная функция *OnDraw* просто рисует пузырьки в окне представления. Добавьте выделенный код:

```
void CEx17bView::OnDraw(CDC* pDC)
{
    int i, j;

    CEx17bDoc* pDoc = GetDocument();
    j = pDoc->m_ellipseArray.GetUpperBound();
    for (i = 0; i < j; i++) {
        pDC->Ellipse(pDoc->m_ellipseArray[i]);
    }
}
```

7. **Вставьте функцию *OnPrepareDC* в Ex17bView.cpp.** Класс «вид» не предусматривает прокрутки в окне представления, поэтому режим преобразования ко-

ординат надо установить именно в этой функции. В окне Properties утилиты Class View переопределите функцию *OnPrepareDC* и введите выделенный код:

```
void CEx17bView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetMapMode(MM_LOENGLISH);
}
```

8. **Вставьте функцию *OnPrint* в Ex17bView.cpp.** Исходная функция *CView::OnPrint* вызывает *OnDraw*. Мы собираемся печатать информацию, отличную от выведенной на экране, поэтому в *OnPrint* надо обойтись без вызова *OnDraw*. Функция *OnPrint* сначала устанавливает режим *MM_TWIPS*, потом создает шрифт фиксированной ширины. После распечатки числовых значений 12 элементов *m_ellipseArray* шрифт отключается. Можно было бы создать шрифт лишь раз в *OnBeginPrinting*, но в данном случае это не дало бы заметного выигрыша. В окне Properties утилиты Class View переопределите функцию *OnPrint*, а затем добавьте в нее выделенный код:

```
void CEx17bView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    int     i, nStart, nEnd, nHeight;
    CString   str;
    CPoint  point(720, -1440);
    CFont    font;
    TEXTMETRIC tm;

    pDC->SetMapMode(MM_TWIPS);
    CEx17bDoc* pDoc = GetDocument();
    m_nPage = pInfo->m_nCurPage; // для функции PrintPageFooter
    nStart = (m_nPage - 1) * CEx17bDoc::nLinesPerPage;
    nEnd = nStart + CEx17bDoc::nLinesPerPage;
     // фиксированный шрифт размером 14 пт
    font.CreateFont(-280, 0, 0, 0, 400, FALSE, FALSE,
                 0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                 CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                 DEFAULT_PITCH | FF_MODERN, "Courier New");
                 // Courier New - это TrueType-шрифт
    CFont* pOldFont = (CFont*) (pDC->SelectObject(&font));
    PrintPageHeader(pDC);
    pDC->GetTextMetrics(&tm);
    nHeight = tm.tmHeight + tm.tmExternalLeading;
    for (i = nStart; i < nEnd; i++) {
        if (i > pDoc->m_ellipseArray.GetUpperBound()) {
            break;
        }
        str.Format("%6d %6d %6d %6d %6d", i + 1,
                pDoc->m_ellipseArray[i].left,
                pDoc->m_ellipseArray[i].top,
                pDoc->m_ellipseArray[i].right,
                pDoc->m_ellipseArray[i].bottom);
        point.y -= nHeight;
```

```
        pDC->TextOut(point.x, point.y, str);
    }
    PrintPageFooter(pDC);
    pDC->SelectObject(pOldFont);
}
```

9. **Отредактируйте функцию *OnPreparePrinting* в Ex17bView.cpp.** Эта функция, шаблон которой генерирует MFC Application Wizard, вычисляет количество страниц в документе и — посредством функции *SetMaxPage* — сообщает результат каркасу приложений. Добавьте выделенный код:

```
BOOL CEx17bView::OnPreparePrinting(CPrintInfo* pInfo)
{
    CEx17bDoc* pDoc = GetDocument();
    pInfo->SetMaxPage(pDoc->m_ellipseArray.GetUpperBound() /
                  CEx17bDoc::nLinesPerPage + 1);
    return DoPreparePrinting(pInfo);
}
```

10. **Вставьте в Ex17bView.cpp функции, формирующие колонтитулы.** Эти закрытые функции, вызываемые из *OnPrint*, печатают верхние и нижние колонтитулы. Нижний колонтитул содержит номер страницы, сохраненный *OnPrint* в переменной-члене *m_nPage* класса «вид». Функция *CDC::GetTextExtent* вычисляет ширину строки с номером страницы, чтобы ее можно было выровнять по правому полю. Добавьте выделенный код:

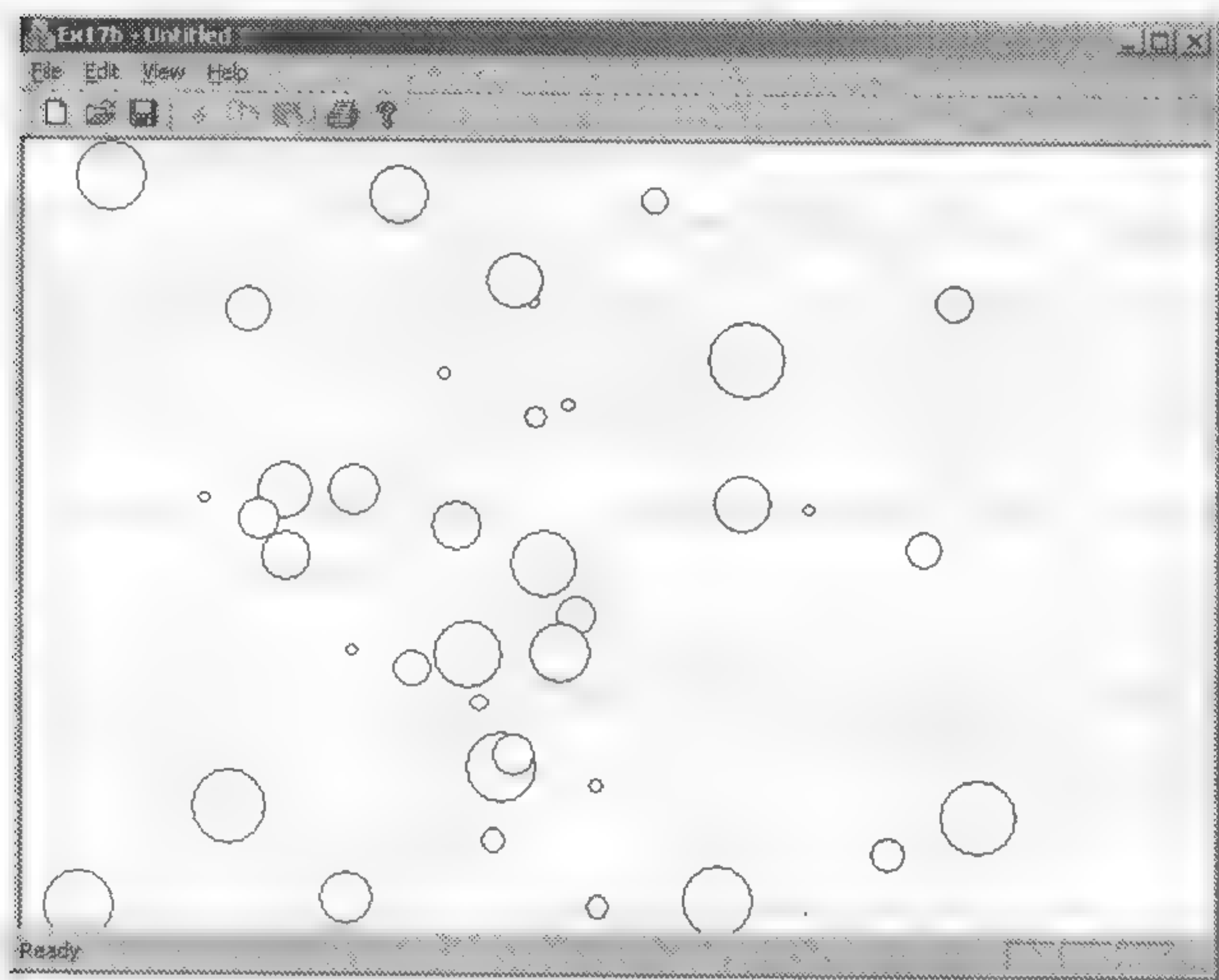
```
void CEx17bView::PrintPageHeader(CDC* pDC)
{
    CString str;

    CPoint point(0, 0);
    pDC->TextOut(point.x, point.y, "Bubble Report");
    point += CSize(720, -720);
    str.Format("%6.6s %6.6s %6.6s %6.6s %6.6s",
            "Index", "Left", "Top", "Right", "Bottom");
    pDC->TextOut(point.x, point.y, str);
}

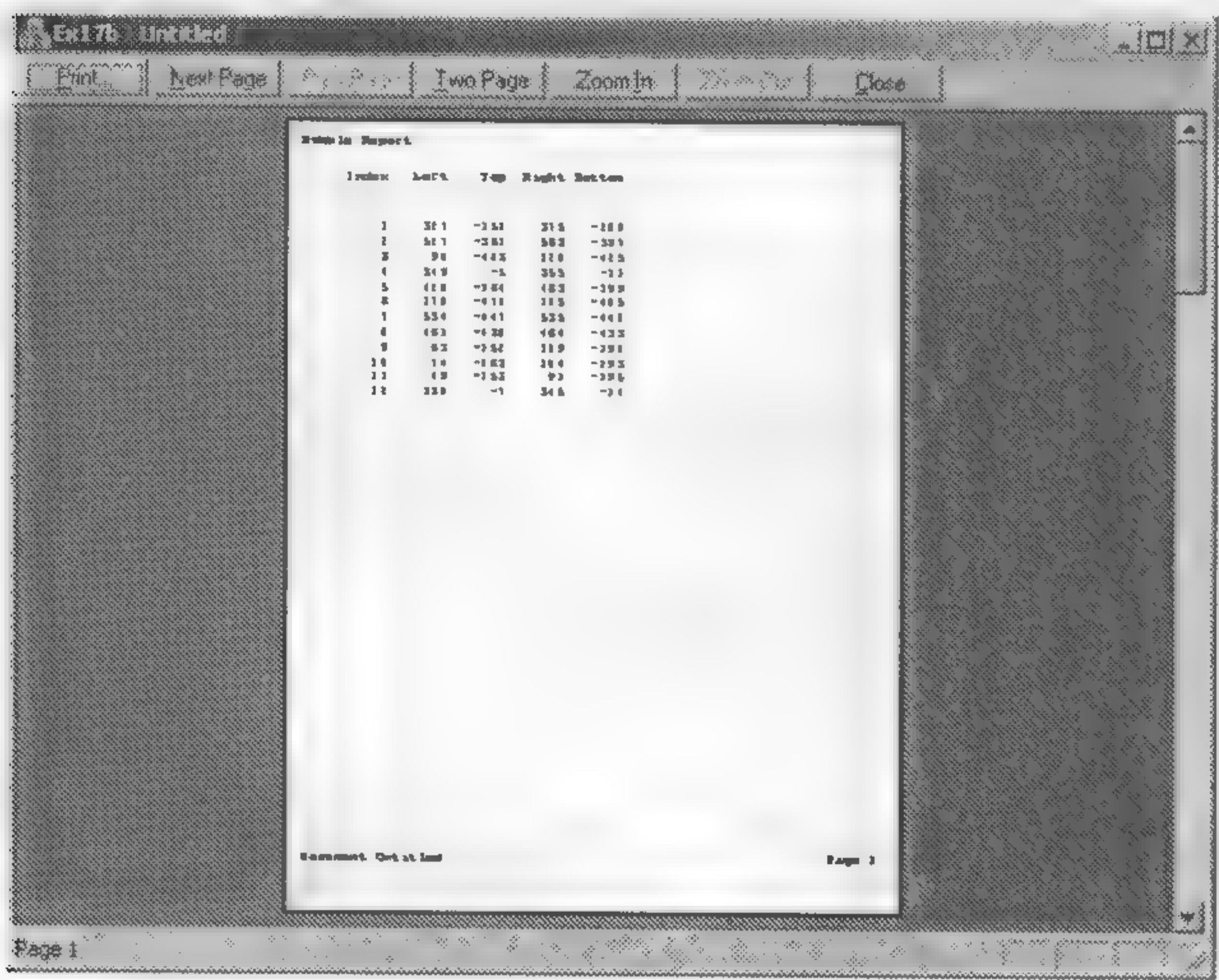
void CEx17bView::PrintPageFooter(CDC* pDC)
{
    CString str;

    CPoint point(0, -14400); // Смещает на 10 дюймов вниз
    CEx17bDoc* pDoc = GetDocument();
    str.Format("Document %s", (LPCSTR) pDoc->GetTitle());
    pDC->TextOut(point.x, point.y, str);
    str.Format("Page %d", m_nPage);
    CSize size = pDC->GetTextExtent(str);
    point.x += 11520 - size.cx;
    pDC->TextOut(point.x, point.y, str); // выравнивание по правому краю
}
```

11. **Соберите и протестируйте приложение.** Запустив программу, вы увидите нечто вроде:



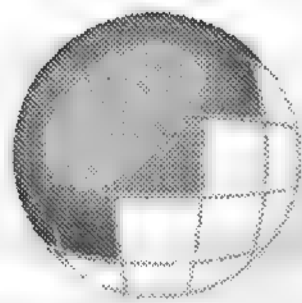
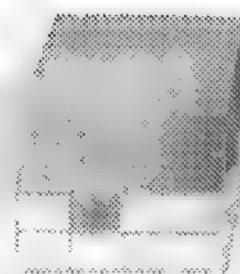
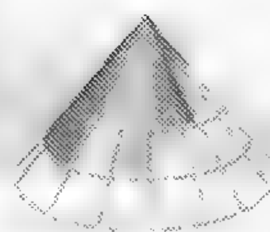
При каждом выборе команды New из меню File изображение на экране должно меняться, а в режиме Print Preview первая страница должна выглядеть так:



В диалоговом окне Print можно задать диапазон печатаемых страниц.

ГЛАВА

18

Разделяемые окна и множественное представление данных

Во всех программах, с которыми вы имели дело до сих пор, кроме Ex16b, было только одно связанное с документом окно представления. Если вы работали в каком-нибудь текстовом процессоре для Windows, то знаете, насколько удобно, когда разные части одного документа открыты сразу в двух окнах. Оба окна могут показывать документ в обычном виде, а могут, скажем, и так: одно окно находится в режиме разметки страницы, а другое — в режиме структуры.

Каркас приложений позволяет реализовать множественное представление документа несколькими способами, в том числе *разделением окна на секции* (splitter window) и созданием нескольких дочерних MDI-окон. В этой главе мы рассмотрим оба варианта, и вы увидите, что сформировать множество *объектов* одного класса «вид» (обычное представление документа) не так уж трудно. Немного сложнее использовать в одном и том же приложении два или более *класса* «вид» (например, виды структуры и разметки страницы).

В этой главе мы сделаем упор на создании нескольких представлений. В примерах предполагается, что данные документа инициализируются в функции *OnNewDocument*. Советуем обратиться к главе 15, чтобы освежить знания о взаимодействии «документ-вид».

Разделяемое окно

Это окно — особая разновидность окна-рамки; каждая его *секция* (pane) содержит свое представление документа. Разбить окно на секции может сама программа при его создании или пользователь, выбрав соответствующую команду меню

либо переместив маркер разбиения на полосе прокрутки. После того как окно разбито на секции, маркер разбиения позволяет корректировать размеры секций (для этого его перемещают мышью). Разделяемые окна применяются как в SDI-, так и в MDI-приложениях.

Разделяемое окно — объект класса *CSplitterWnd* — полностью занимает клиентскую область окна-рамки (класса *CFrameWnd* или *CMDIChildWnd*), а в его секциях располагаются окна представления. Разделяемое окно не участвует в маршрутизации команд. Активное окно представления (в секции разделяемого окна) логически связано напрямую с окном-рамкой.

Варианты создания множественных представлений

Реализовать множественное представление с моделями приложений можно несколькими способами.

- **SDI-приложение с разделяемым окном, один класс «вид».** Этот вариант реализован в программе Ex18a. Каждая секция независимо от других позволяет прокручивать содержимое документа до любого места. Программист задает максимальное число горизонтальных и вертикальных секций, а пользователь при работе с приложением сам разделяет окно.
- **SDI-приложение с разделяемым окном, несколько классов «вид».** Этот вариант показан в примере Ex18b. Программист задает число секций и последовательность формирования объектов «вид», а пользователь при работе с приложением может изменять размеры секций.
- **SDI-приложение без разделяемых окон, несколько классов «вид».** Этот вариант воплощен в программе Ex18c. Пользователь переключает классы «вид», выбирая соответствующие команды меню.
- **MDI-приложение без разделяемых окон, один класс «вид».** Это стандартное MDI-приложение вы видели в главе 16. Командой New Window можно было создать новое дочернее окно для уже открытого документа.
- **MDI-приложение без разделяемых окон, несколько классов «вид».** Это разновидность стандартного MDI-приложения, позволяющая работать с множественным представлением документа. Как показано в примере Ex18d, для этого нужно лишь заменить New Window командами, соответствующими каждому из доступных в программе классов «вид».
- **MDI-приложение с разделяемыми дочерними окнами.** Этот вариант подробно рассмотрен в электронной документации «MFC Library Reference» на примере программы SCRIBBLE.

Динамически и статически разделяемые окна

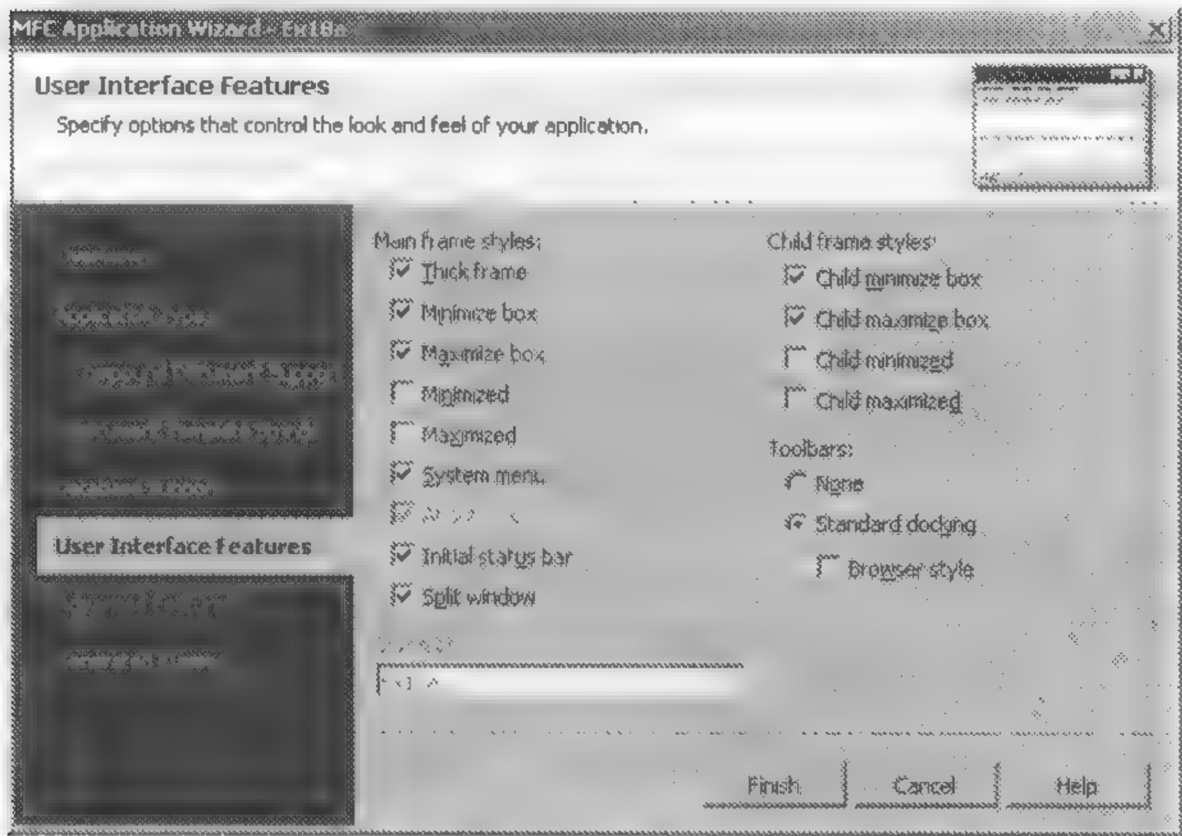
Динамически разделяемое окно (dynamic splitter window) можно разделять в любой момент, выбирая соответствующую команду или перемещая маркер разбиения на полосе прокрутки. Секции в динамически разделяемом окне обычно связаны с одним классом «вид». Верхняя левая секция показывает выбранное пред-

ставление данных сразу после создания разделяемого окна. Полосы прокрутки — общие для всех секций. Например, в окне, разбитом по горизонтали на две секции, нижняя полоса прокрутки управляет обоими окнами представления. При запуске приложения с динамически разделяемым окном формируется единственный объект «вид». Когда пользователь разделяет окно-рамку, создаются другие объекты «вид», а когда «воссоединяет» его, эти объекты уничтожаются.

Секции *статически разделяемого окна* (static splitter window) определяются при создании окна; пользователь вправе изменять их размеры, но не число. Статически разделяемые окна подходят для нескольких классов «вид»; при этом конфигурация окон устанавливается при их создании. В статически разделяемом окне у каждой секции свои полосы прокрутки. В приложении со статически разделяемым окном все объекты «вид» конструируются при создании окна-рамки и удаляются при его уничтожении.

Пример Ex18a: SDI-приложение с динамически разделяемым окном и одним классом «вид»

В этом примере окно можно динамически разделить на четыре секции, что приводит к формированию четырех объектов «вид», и всеми управляет один класс «вид». Код классов «документ» и «вид» мы возьмем из примера Ex17a. Добавить динамически разделяемое окно к новому приложению поможет MFC Application Wizard: создайте SDI-приложение и на странице User Interface Features установите флажок Split window (использовать разделяемое окно):



При этом MFC Application Wizard добавит код к классу *CMainFrame*. Разумеется, к существующему приложению такой код придется добавить вручную.

Ресурсы для разделения окна

Генерируя приложение с разделяемым окном-рамкой, MFC Application Wizard дополняет меню View проекта командой Split. Идентификатор команды, *ID_WINDOW_SPLIT*, соотносится с функцией-обработчиком класса *CView* из библиотеки MFC.

CMainFrame

Классу основного окна-рамки приложения нужна переменная-член, связанная с разделяемым окном, и прототип переопределенной функции *OnCreateClient*. Следующий код MFC Application Wizard добавит в файл MainFrm.h:

```
protected:
    CSplitterWnd m_wndSplitter;
public:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);
```

При создании объекта-рамки каркас приложений вызывает виртуальную функцию-член *CFrameWnd::OnCreateClient*. Базовый класс формирует единственное окно представления, определенное в шаблоне документа. Функция *OnCreateClient*, переопределенная мастером MFC Application Wizard в MainFrm.cpp (см. ниже), создает вместо этого разделяемое окно, а то в свою очередь формирует первое окно представления:

```
BOOL CMainFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    return m_wndSplitter.Create( this,
        2, 2,               // TODO: adjust the number of rows, columns
        CSize(10, 10),      // TODO: adjust the minimum pane size
        pContext);
}
```

Функция-член *CSplitterWnd::Create* создает динамически разделяемое окно. Класс «вид» известен объекту *CSplitterWnd*, поскольку имя этого класса внедрено в структуру *CCreateContext*, передаваемую в *Create* как параметр.

Второй и третий параметры функции *Create* (*2, 2*) указывают, что окно можно разбить максимум на две строки и на два столбца (т. е. на 4 секции). Заменив (*2, 2*) на (*2, 1*), вы получите две горизонтальные секции, а (*1, 2*) дадут две вертикальные. Параметр *CSize* определяет минимальный размер секции.

Тестирование приложения Ex18a

После запуска Ex18a окно можно разделить командой Split меню View или маркерами разбиения на полосах прокрутки. Ниже показано окно представления, разделенное на 4 секции (рис. 18-1). На все представления приходится один набор полос прокрутки.

Пример Ex18b: SDI-приложение с статически разделяемым окном и двумя классами «вид»

Ex18b — расширенный вариант программы Ex18a; в ней определен второй класс «вид», благодаря чему в статически разделяемом окне располагаются два представления одного документа. (CPP- и H-файлы взяты от первоначального класса «вид».) На этот раз разделяемое окно ведет себя чуть иначе. При запуске сразу форми-

руются две секции, и пользователь вправе варьировать их размер маркером разбиения на правой полосе прокрутки или командой Split из меню Window.

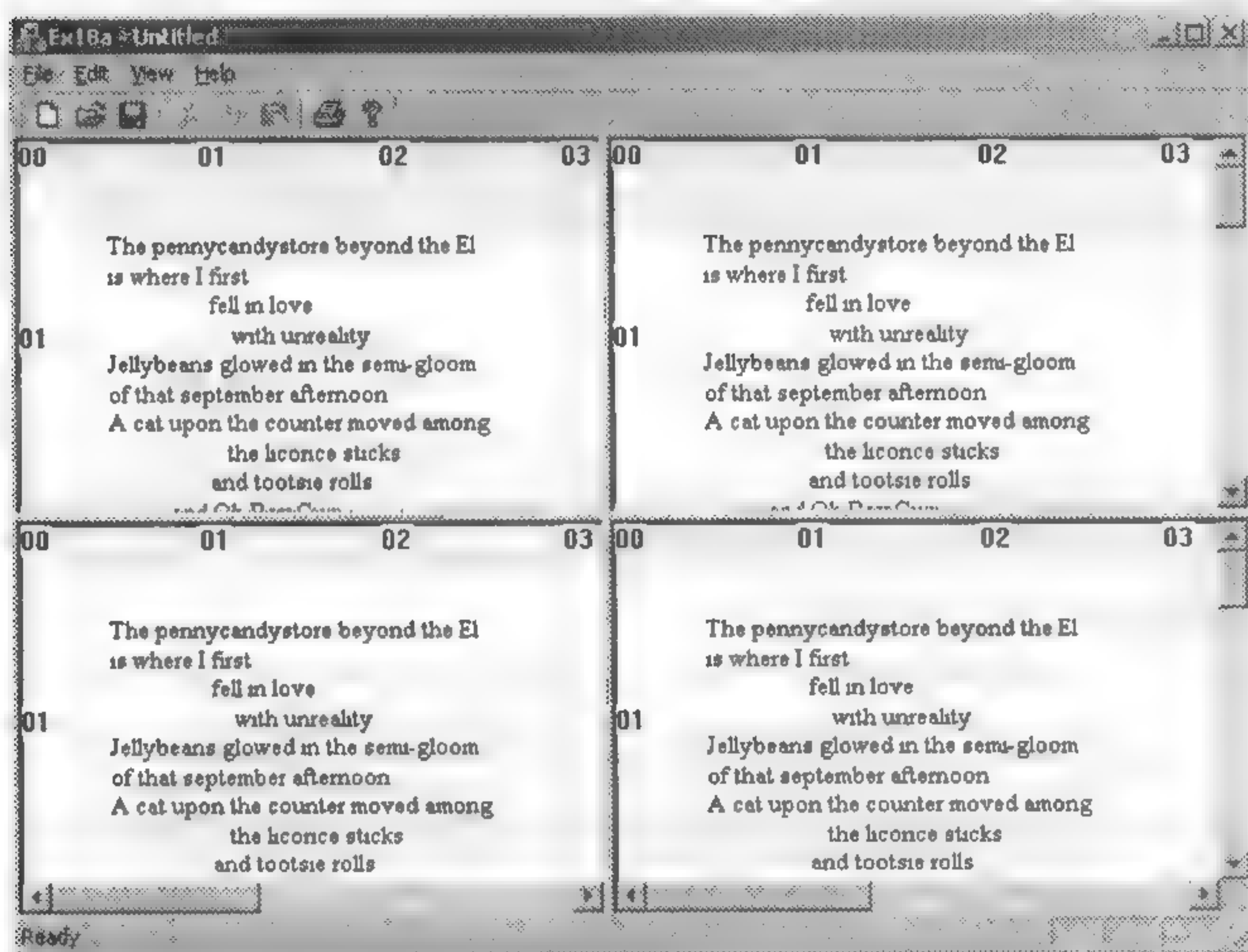


Рис. 18-1. *Единственное окно представления, разделенное на четыре секции*

Простейший способ создать приложение со статически разделяемым окном — сгенерировать его с динамически разделяемым окном с помощью MFC Application Wizard и отредактировать функцию *CMainFrame::OnCreateClient.*

CHexView

Класс *CHexView* адресован любителям поэзии в шестнадцатеричном виде. Он отличается от *CStringView* только функцией-членом *OnDraw*:

```
void CHexView::OnDraw(CDC* pDC)
{
   // шестнадцатеричный дамп строк документа
   int          i, j, k, l, n, nHeight;
   CString      outputLine, str;
   CFont        font;
   TEXTMETRIC   tm;

   CPoemDoc* pDoc = GetDocument();
   font.CreateFont(-160, 80, 0, 0, 400, FALSE, FALSE, 0, ANSI_CHARSET,
         OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
         DEFAULT_PITCH | FF_SWISS, "Arial" );
   CFont* pOldFont = pDC->SelectObject(&font);
   pDC->GetTextMetrics(&tm);
   nHeight = tm.tmHeight + tm.tmExternalLeading;

   j = pDoc->m_stringArray.GetSize();
   for (i = 0; i < j; i++ ) {
      outputLine.Format("%02x ", i);
```

```
        l = pDoc->m_stringArray[i].GetLength();
        for (k = 0; k < l; k++ ) {
            n = pDoc->m_stringArray[i][k] & 0x00ff;
            str.Format("%02x ", n);
            outputLine += str;
        }
        pDC->TextOut(720, -i * nHeight - 720, outputLine);
    }
    pDC->SelectObject(pOldFont);
}
```

Эта функция показывает шестнадцатеричный дамп всех строк в наборе *m_stringArray* документа. Обратите внимание на оператор индексации, используемый для доступа к отдельным символам в объекте *CString*.

CMainFrame

Как и в Ex18a, классу основного окна-рамки в Ex18b нужна переменная-член, связанная с разделяемым окном, и прототип переопределенной функции *OnCreateClient*. Чтобы сгенерировать код, используя MFC Application Wizard, установите флажок Split window (как в Ex18a). При этом модифицировать файл MainFrm.h не придется.

Для файла реализации (MainFrm.cpp) нужны заголовочные файлы классов «документ» и «вид»:

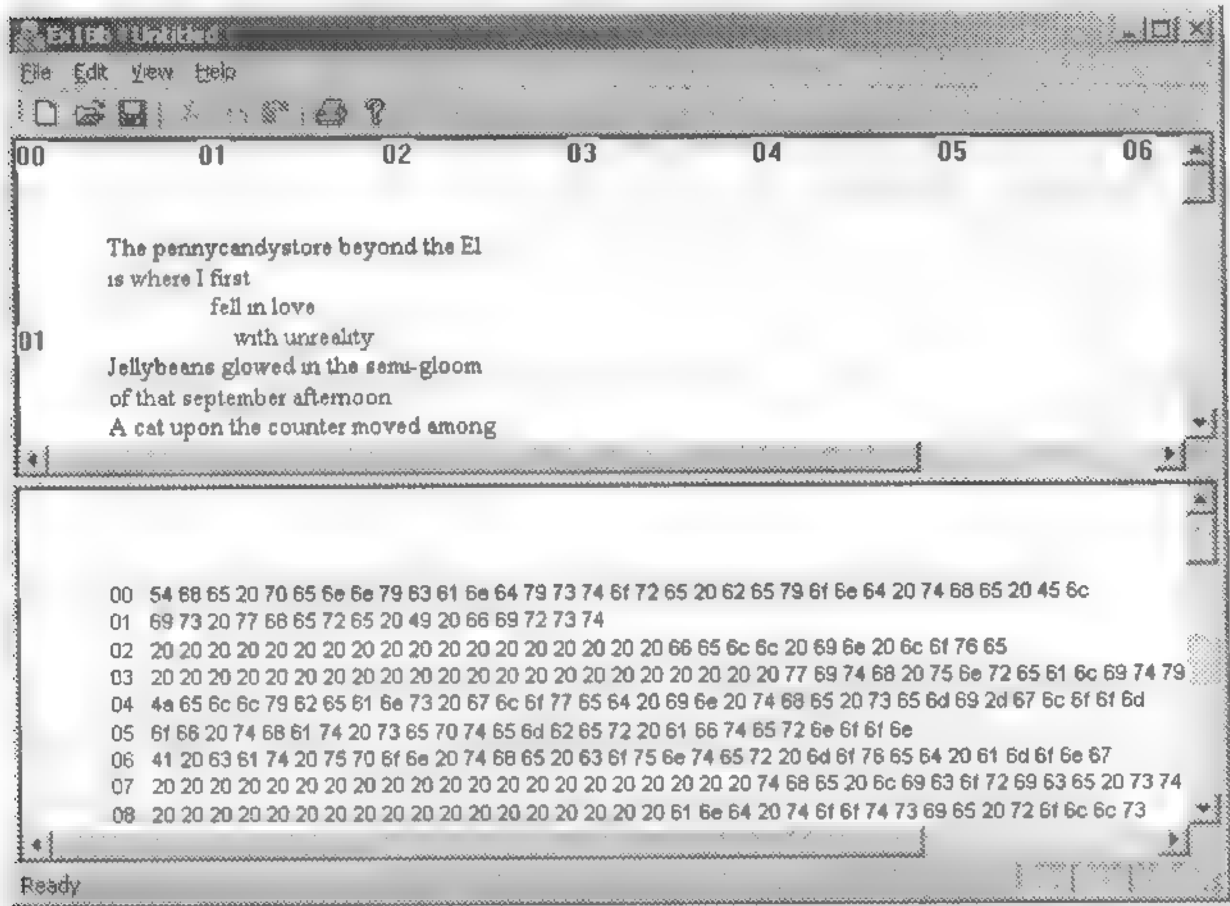
```
#include "PoemDoc.h"
#include "StringView.h"
#include "HexView.h"
```

MFC Application Wizard генерирует в функции *OnCreateClient* код динамически разделяемого окна, поэтому, чтобы сформировать статически разделяемое окно, эту функцию придется подправить. Вместо *CSplitterWnd::Create* вы должны вызвать функцию *CSplitterWnd::CreateStatic* — она предназначена для работы с несколькими классами «вид». Последующие вызовы *CSplitterWnd::CreateView* подключают два таких класса. Последние два параметра (*2*, *1*) функции *CreateStatic* заставляют формировать разделяемое окно с двумя секциями; при этом в начальном состоянии разделительная линия отстоит от верхней части окна на 100 единиц (в аппаратных координатах). Верхняя секция — это обычное (текстовое) представление документа, а нижняя — его шестнадцатеричный дамп. Положение разделительной линии изменять можно, а конфигурацию представления — нет.

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/, CCreateContext* pContext)
{
    VERIFY(m_wndSplitter.CreateStatic(this, 2, 1));
    VERIFY(m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CStringView),
        CSize(100, 100), pContext));
    VERIFY(m_wndSplitter.CreateView(1, 0, RUNTIME_CLASS(CHexView),
        CSize(100, 100), pContext));
    return TRUE;
}
```

Тестирование приложения Ex18b

После запуска окно программы Ex18b будет выглядеть, как показано на рисунке. Обратите внимание: в каждой секции свои горизонтальные полосы прокрутки.



Пример Ex18c: переключение между классами «вид» без разделения окна

Иногда надо программно переключаться между классами «вид», а связываться с разделяемым окном не хочется. Так вот, пример Ex18c — это SDI-приложение, которое переключается между *CStringView* и *CHexView* в ответ на команды из меню View. При создании приложения средствами MFC Application Wizard вам требуется лишь добавить две новые команды в меню и немного кода в класс *CMainFrame*. А еще сделать ранее защищенные конструкторы *CStringView* и *CHexView* открытыми.

Требования к ресурсам

В ресурс меню *IDR_MAINFRAME* к меню View добавлены два элемента.

Элемент меню	Идентификатор команды	CMainFrame-функция
St&ring View	*ID_VIEW_STRINGVIEW*	*OnViewStringView*
&Hex View	*ID_VIEW_HEXVIEW*	*OnViewHexView*

Функции-обработчики команд (и обработчики обновления командного пользовательского интерфейса) добавляются в класс *CMainFrame* в окне Properties утилиты Class View.

CMainFrame

Класс *CMainFrame* дополняется закрытой вспомогательной функцией *SwitchToView*, вызываемой из двух обработчиков команд меню. Параметр перечислимого типа сообщает функции, на какой из объектов «вид» ей переключиться. Следующий код надо добавить в заголовочный файл MainFrm.h:

```
private:
    enum eView {STRING = 1, HEX = 2};
    void SwitchToView(eView nView);
```

Чтобы найти и активизировать запрошенный объект «вид», функция *SwitchToView* (в MainFrm.cpp), делает ряд низкоуровневых вызовов MFC-функций. Не забивайте себе голову тем, как все это работает, — вы всегда сможете адаптировать эту функцию к своей программе. Добавьте этот код:

```
void CMainFrame::SwitchToView(eView nView)
{
    CView* pOldActiveView = GetActiveView();
    CView* pNewActiveView = (CView*) GetDlgItem(nView);
    if (pNewActiveView == NULL) {
        switch (nView) {
        case STRING:
            pNewActiveView = (CView*) new CStringView;
            break;
        case HEX:
            pNewActiveView = (CView*) new CHexView;
            break;
        }
        CCreateContext context;
        context.m_pCurrentDoc = pOldActiveView->GetDocument();
        pNewActiveView->Create(NULL, NULL, WS_BORDER,
            CFrameWnd::rectDefault, this, nView, &context);
        pNewActiveView->OnInitialUpdate();
    }
    SetActiveView(pNewActiveView);
    pNewActiveView->ShowWindow(SW_SHOW);
    pOldActiveView->ShowWindow(SW_HIDE);
    pOldActiveView->SetDlgCtrlID(
        pOldActiveView->GetRuntimeClass() ==
        RUNTIME_CLASS(CStringView) ? STRING : HEX);
    pNewActiveView->SetDlgCtrlID(AFX_IDW_PANE_FIRST);
    RecalcLayout();
}
```

А теперь рассмотрим обработчики команд меню и обновления командного пользовательского интерфейса, первоначально сгенерированные мастерами окна Class View (вместе с элементами таблицы сообщений и прототипами). Обработчики, обновляющие пользовательский интерфейс, проверяют класс активного объекта «вид».

```
void CMainFrame::OnViewStringView()
{
    SwitchToView(STRING);
}

void CMainFrame::OnUpdateViewStringView(CCmdUI* pCmdUI)
{
```

```
    pCmdUI->Enable(!GetActiveView()->IsKindOf(RUNTIME_CLASS(CStringView)));
}

void CMainFrame::OnViewHexView()
{
    SwitchToView(HEX);
}

void CMainFrame::OnUpdateViewHexView(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!GetActiveView()->IsKindOf(RUNTIME_CLASS(CHexView)));
}
```

Тестирование приложения Ex18c

Ex18c изначально отображает документ в окне представления класса *CStringView*. Вы можете переключаться между *CStringView* и *CHexView*, выбирая соответствующую команду из меню View (рис. 18-2).

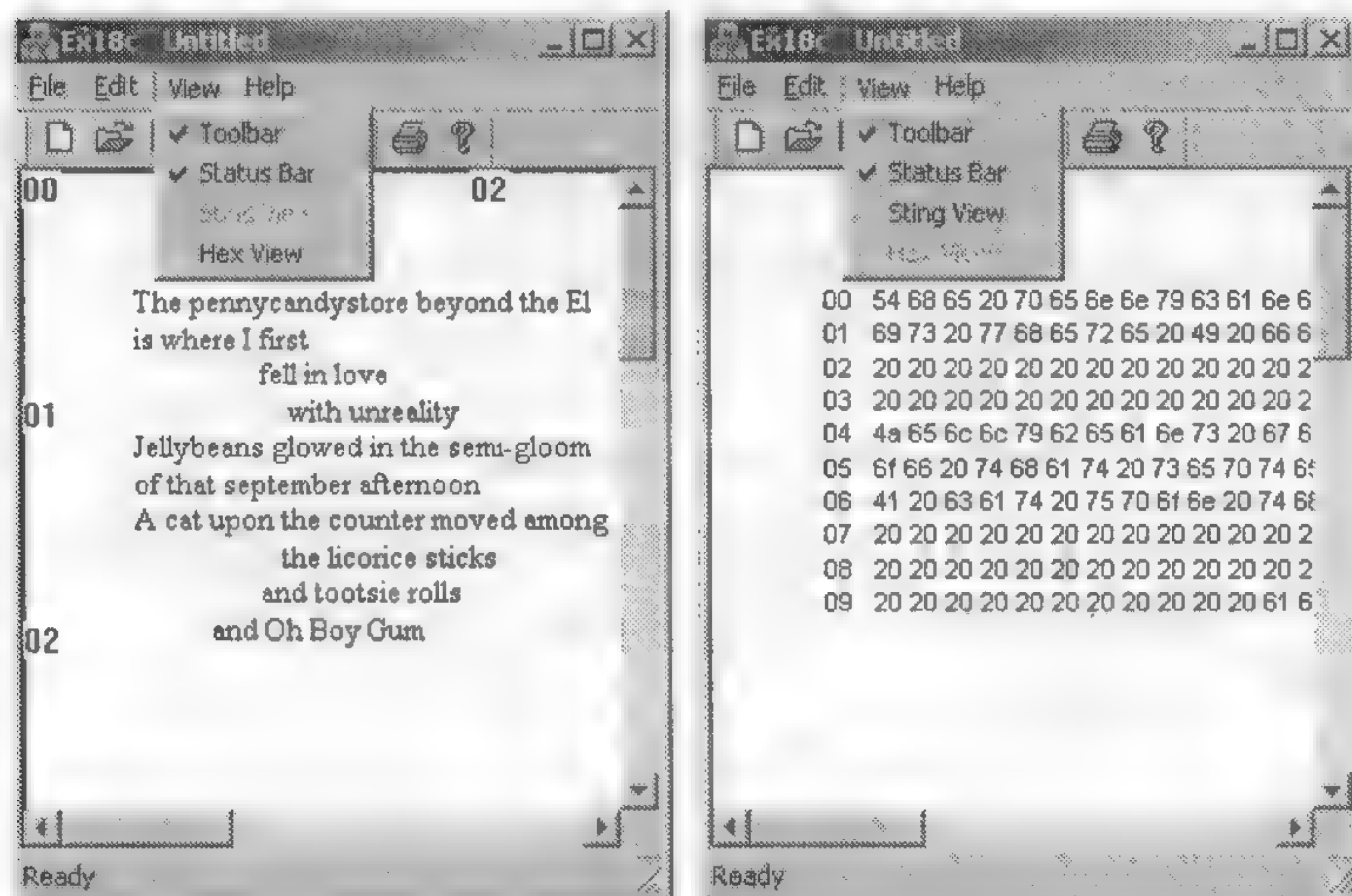


Рис. 18-2. *Представление документа объектами CStringView и CHexView*

Пример Ex18d: MDI-приложение с несколькими классами «вид»

В заключительном примере для создания MDI-приложения с несколькими классами «вид» без разделяемого окна используются предыдущие классы «документ» и «вид». Его логика отличается от логики других программ с несколькими классами «вид». Основные события здесь разворачиваются в классе приложения, а не в классе основного окна-рамки. Изучая Ex18d, вы глубже вникнете в работу с объектами *CDocTemplate*.

Эта программа сгенерирована с установленным флажком Context-sensitive Help на странице Advanced Features мастера MFC Application Wizard. Если вы начинае-

те с нуля, создайте средствами MFC Application Wizard обычное MDI-приложение с одним из классов «вид». Затем добавьте к проекту второй класс «вид» и модифицируйте файлы классов приложения и основного окна-рамки.

Требования к ресурсам

В ресурс меню *IDR_EX18DTYPE* к меню Window добавлены два элемента.

Элемент меню (свойство Caption)	Идентификатор команды	Функция *CMainFrame*
New &String Window (заменяет команду New Window)	*ID_WINDOW_NEWSTRINGWINDOW*	*CMDIFrameWnd::OnWindowNew*
New &Hex Window	*ID_WINDOW_NEWHEXWINDOW*	*OnWindowNewhexWindow*

Обработчик команды *OnWindowNewhex* в классе *CMainFrame* создайте в окне Properties утилиты Class View.

CEx18dApp

В заголовочный файл класса приложения (Ex18d.h) добавьте переменную-член:

```
public:
    CMultiDocTemplate* m_pTemplateHex;
```

Файл реализации Ex18d.cpp должен содержать операторы *#include*:

```
#include "PoemDoc.h"
#include "StringView.h"
#include "HexView.h"
```

В функцию-член *InitInstance* класса *CEx18dApp* вставьте следующий код сразу после вызова функции *AddDocTemplate*:

```
m_pTemplateHex = new CMultiDocTemplate(
    IDR_Ex18dTYPE,
    RUNTIME_CLASS(CPoemDoc),
    RUNTIME_CLASS(CChildFrame),
    RUNTIME_CLASS(CStringView));
```

Вызов *AddDocTemplate*, сгенерированный MFC Application Wizard, устанавливает для приложения первичную комбинацию «документ/рамка/вид», действующую при запуске программы. Показанный выше объект шаблона — это вторичный шаблон, который активизируется в ответ на выбор в меню команды New Hex Window.

Остается создать функцию-член *ExitInstance*, которая очищает вторичный шаблон:

```
int CEx18dApp::ExitInstance()
{
    delete m_pTemplateHex;
    return CWinApp::ExitInstance(); // сохраняет параметры профиля
}
```

CMainFrame

Файл реализации класса основного окна-рамки (MainFrm.cpp) включает заголовочные файлы класса CHexView и класса «документ»:

```
#include "PoemDoc.h"
#include "HexView.h"
```

В базовом классе окна-рамки *CMDIFrameWnd* есть функция *OnWindowNew*, обычно связываемая со стандартной командой New Window меню Window. В Ex18d с этой функцией связана команда меню New String Window, связанная с обработчиком (его текст приведен ниже), который и создает новые дочерние окна с шестнадцатеричным представлением. Эта функция создана на основе *OnWindowNew* и адаптирована для работы со вторичным шаблоном, определенным в *InitInstance*.

```
void CMainFrame::OnWindowNewhexwindow()
{
   CMDIChildWnd* pActiveChild = MDIGetActive();
   CDocument* pDocument;
   if (pActiveChild == NULL ||
         (pDocument = pActiveChild->GetActiveDocument()) == NULL) {
      TRACE("Warning:  No active document for WindowNew command\n");
      AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
      return; // Сбой команды
   }

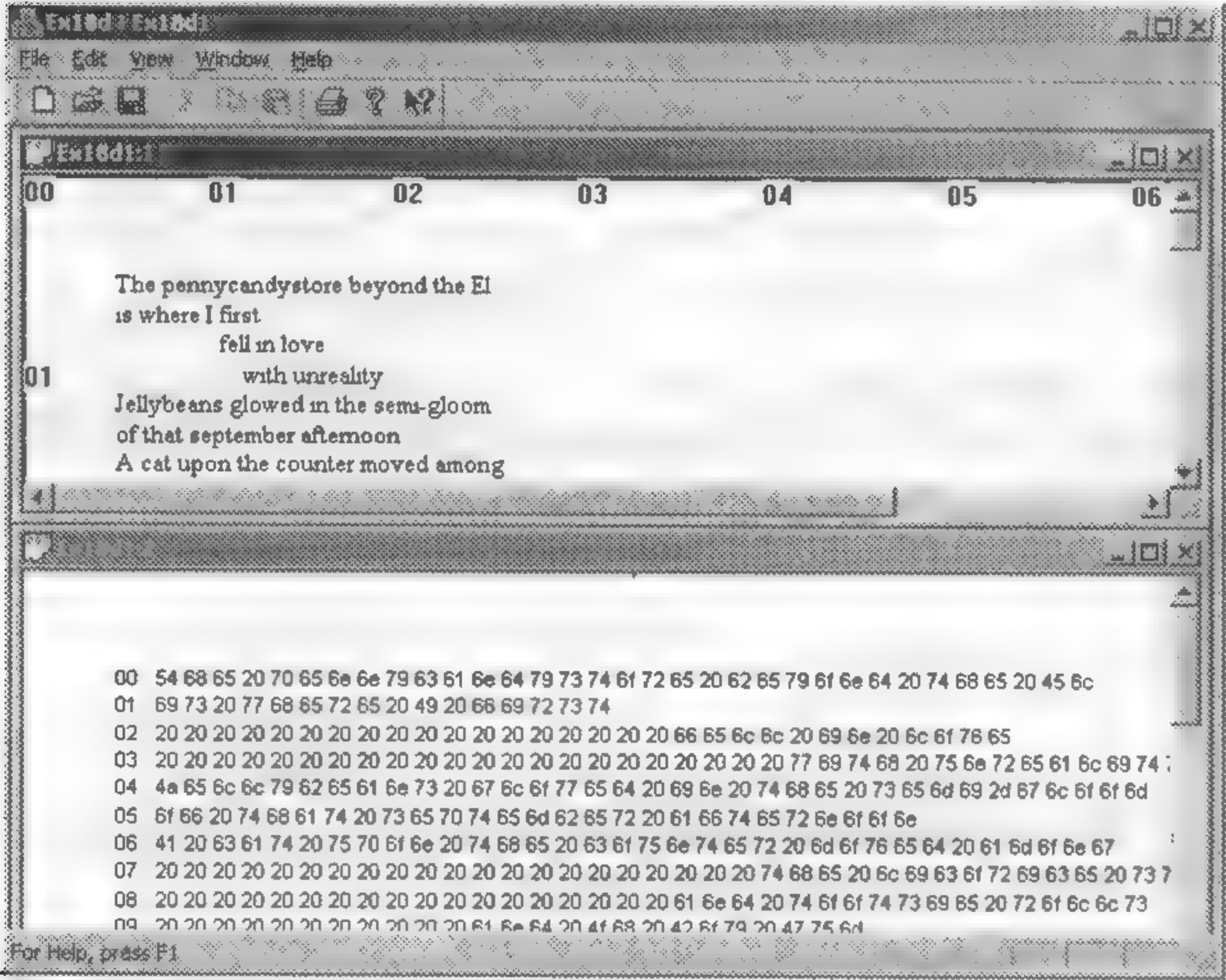
   // В противном случае создается новое окно-рамка!
   CDocTemplate* pTemplate =
      ((CEx18dApp*) AfxGetApp())->m_pTemplateHex;
   ASSERT_VALID(pTemplate);
   CFrameWnd* pFrame =
      pTemplate->CreateNewFrame(pDocument, pActiveChild);
   if (pFrame == NULL) {
      TRACE("Warning:  failed to create new frame\n");
      AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
      return; // Сбой команды
   }

   pTemplate->InitialUpdateFrame(pFrame, pDocument);
}
```

Примечание Продемонстрированное выше «клонирование» функций — очень полезная технология программирования на базе MFC. Ее суть проста. Найдите функцию базового класса, которая делает максимум того, что вам нужно, и скопируйте ее из подкаталога \Vc7\atlmfc\src\mfc в свой производный класс и отредактируйте. Правда, есть риск, что в будущих версиях MFC-библиотеки найденная вами функция окажется реализованной иначе.

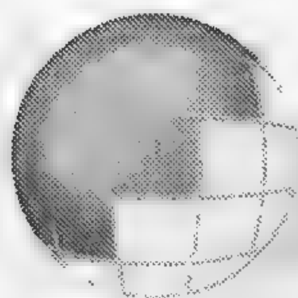
Тестирование приложения Ex18d

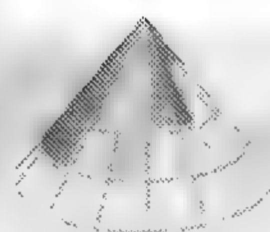
После запуска приложения Ex18d на экране появится дочернее окно с текстовым представлением. Выберите в меню Window команду New Hex Window; у вас должно получиться что-то вроде этого:



ГЛАВА

19





Контекстно-зависимая справка

Сегодня существуют два вида технологии интерактивной справки: в формате HTML (Hypertext Markup Language) и классический формат WinHelp. Программы на основе MFC поддерживают оба вида, но популярность HTML Help неуклонно растет, о чем свидетельствует новая интерактивная документация по Visual C++ .NET в формате HTML Help.

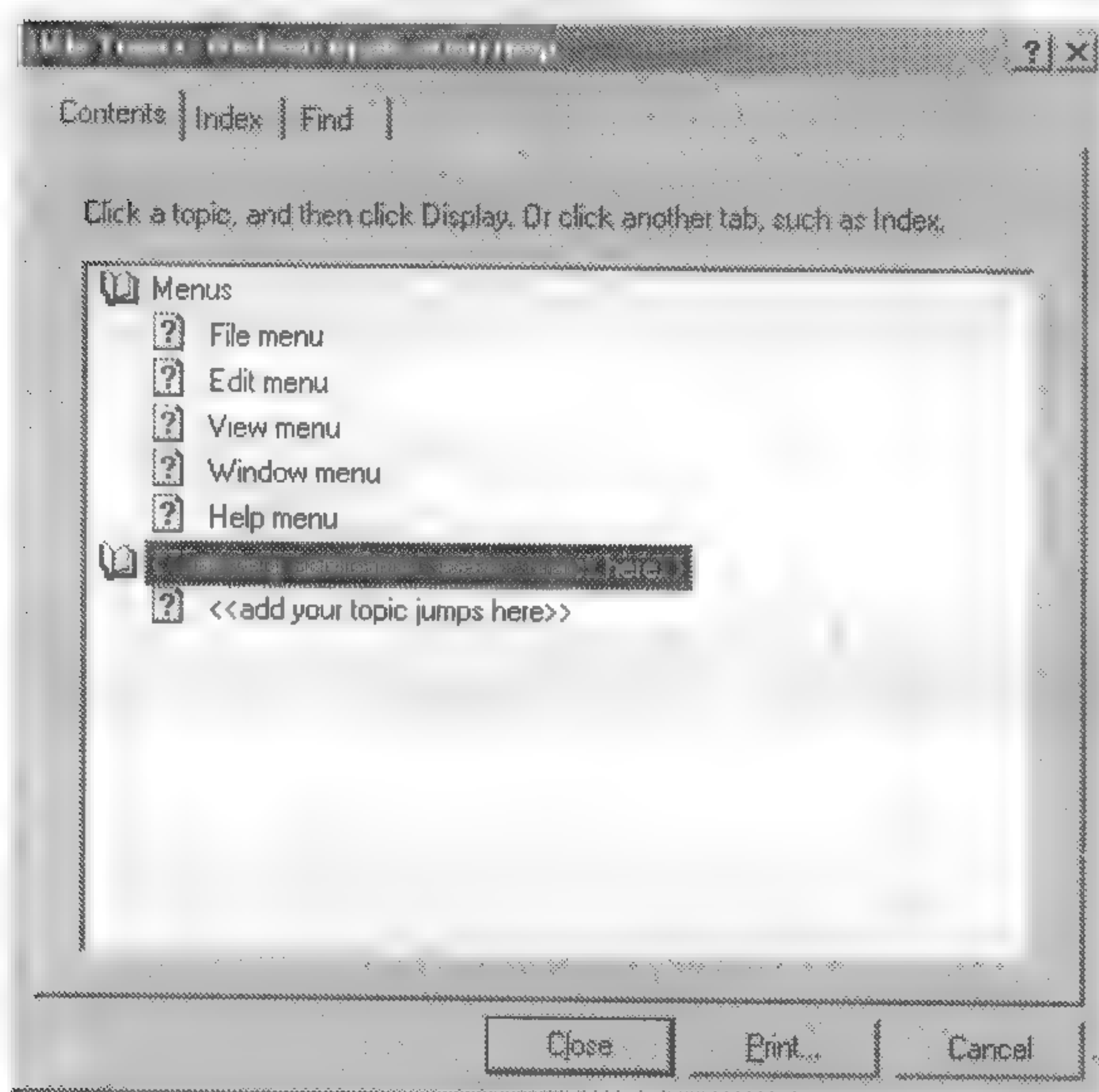
В этой главе мы покажем, как создавать и обрабатывать простой автономный файл справки, содержащий оглавление и поддерживающий переходы между разделами. Далее вы увидите, как программа на основе MFC-библиотеки активизирует справочную систему, передавая ей идентификаторы контекста справки, производные от идентификаторов окна и команд. И, наконец, вы узнаете, как изменить в MFC схему маршрутизации сообщений справочной системы в соответствии со своими потребностями.

WinHelp и HTML Help

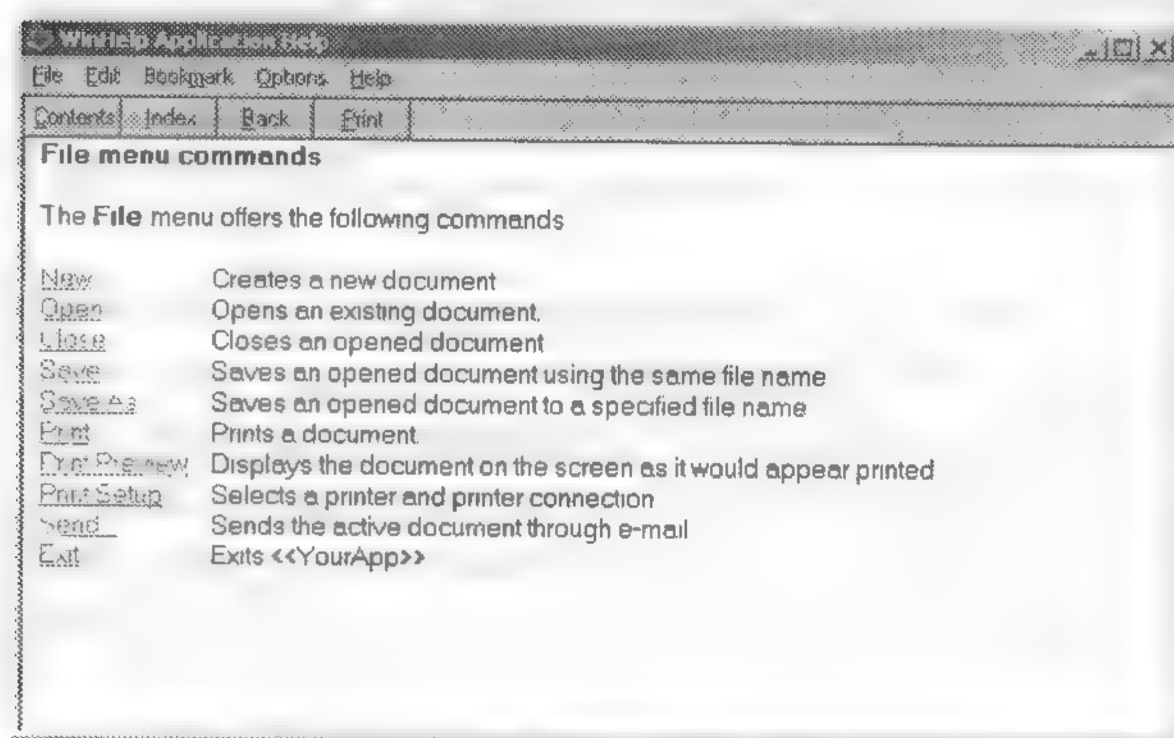
Вообще выбор между WinHelp и HTML Help — вопрос личных предпочтений. Программный интерфейс доступа и управления обеими справочными системами одинаков. В WinHelp применяется текст с форматированием (Rich Text Format, RTF), а в HTML Help — HTML-текст. В последнее время появилось множество коммерческих средств создания справочных систем для Windows, в том числе RoboHELP от Blue Sky Software и ForeHelp от Forefront Corporation, которые превратили создание справочных систем в формате WinHelp в простую задачу. Однако WinHelp, по-видимому, со временем уступит дорогу HTML Help.

В классической WinHelp-системе доступ к разделам последовательный: доступ к разделам-темам осуществляется через предметный указатель (index) или оглав-

ление. Выбранный раздел WinHelp открывает в отдельном окне. Вот пример окна справочной системы WinHelp, по умолчанию создаваемой MFC Application Wizard:



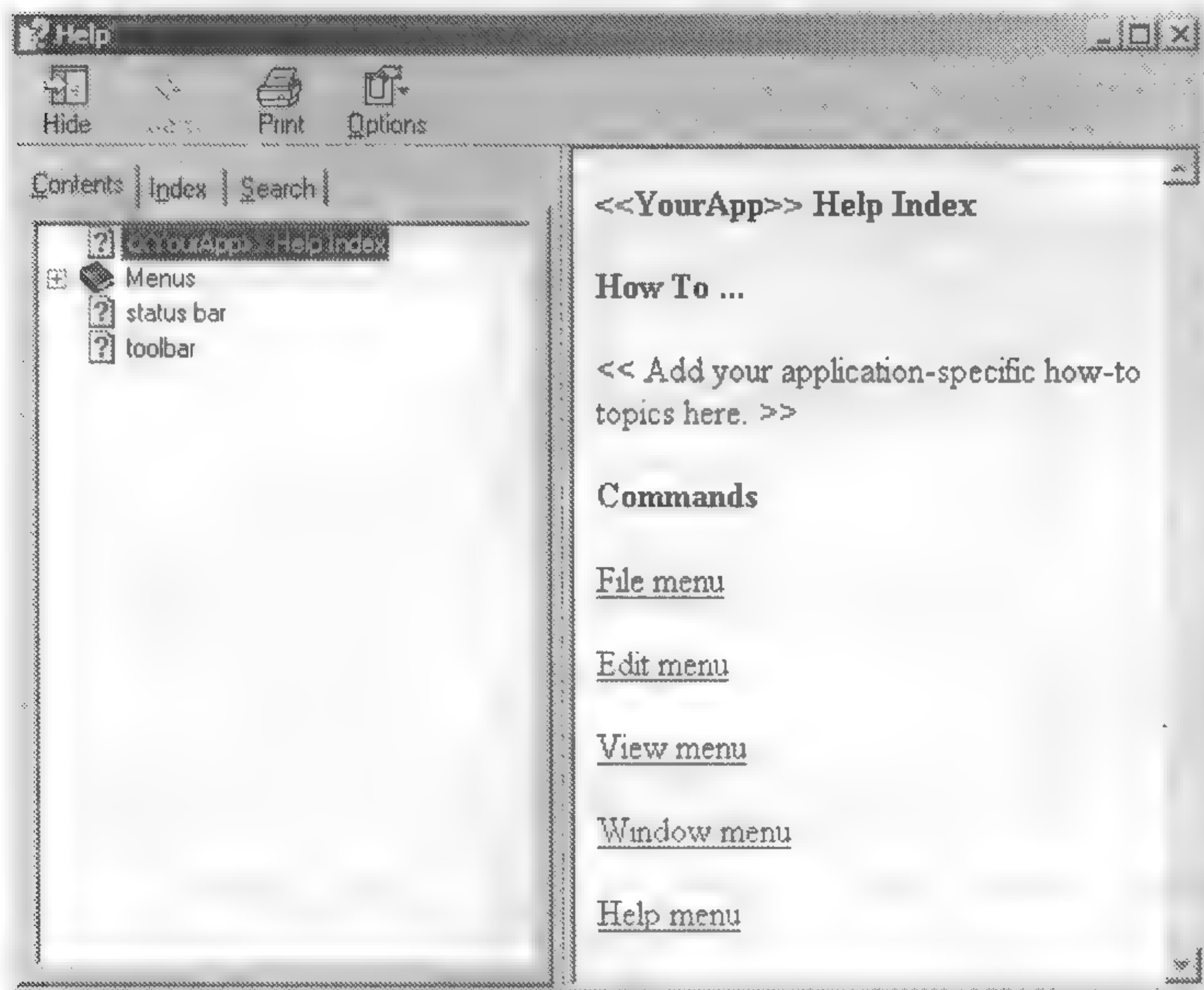
А это окно с разделом «File menu commands». Для перехода назад в оглавление или в предметный указатель служит соответствующая кнопка.



А на следующей странийе показан пример окна справочной системы HTML Help, которое по умолчанию создает MFC Application Wizard: в левой панели окна — вкладки Index (предметный указатель), Contents (оглавление) и Search (поиск), а в правой — содержимое раздела.

HTML Help реализована на основе ActiveX-элемента управления HHCtrl.ocx, который отвечает за навигацию и управление дополнительными и всплывающими окнами. HHCtrl.ocx достаточно гибок, чтобы отображать и скомпилированные файлы справки, и HTML-страницы, которые обычно отображаются в Web-браузере.

Сначала мы познакомимся с работой WinHelp в MFC-приложении.



Windows-программа с WinHelp

Если вы работали с коммерческими Windows-приложениями, то, наверное, обращали внимание на их справочные системы со сложными страницами с графикой, гиперссылками и всплывающими окнами. Создание справочных систем стало профессией. Эта глава не сделает вас экспертом по справочным системам, но позволит с чего-то начать и подготовить простой справочный файл без особых «наворотов».

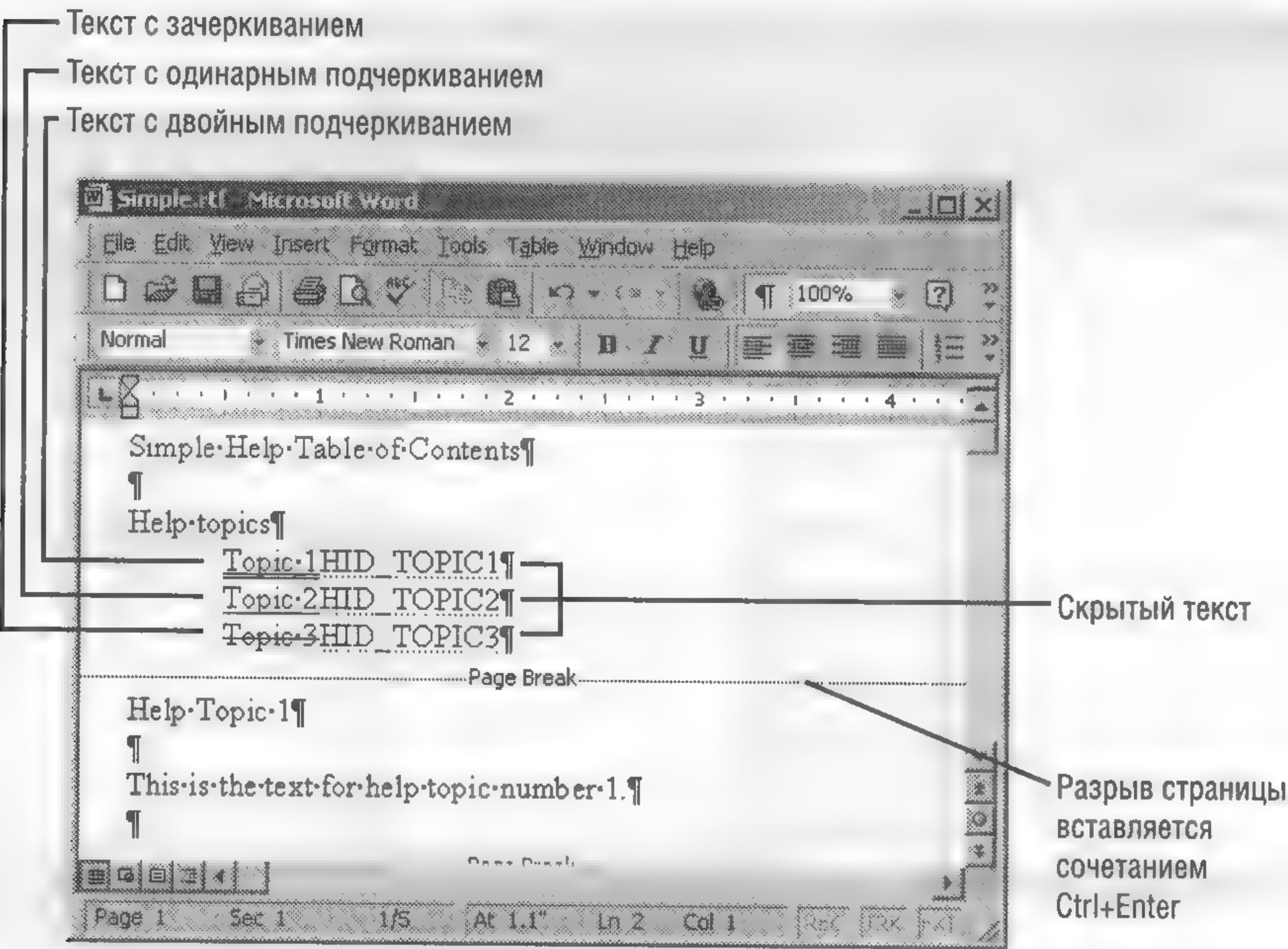
Текст с форматированием

В Windows SDK рассказывается, как создавать текстовые ASCII-файлы справки в стандарте RTF. Мы тоже воспользуемся этим RTF-форматом, но будем работать в режиме WYSIWYG, избегая прямого обращения к громоздким последовательностям управляющих символов (Esc-последовательностям). Будем применять в справочном файле те же шрифты (тех же размеров и начертания), которые пользователь нашей программы увидит в окнах справочной системы. Естественно, нам понадобится текстовый редактор, работающий с форматом RTF. Лично мы предпочитаем Microsoft Word, но формат RTF понимают и многие другие текстовые редакторы.

Подготовка простого справочного файла

Давайте напишем простой справочный файл с оглавлением и тремя тематическими разделами. Этот файл предназначен для чтения напрямую из WinHelp. На C++ ничего программировать не надо.

1. **Создайте подкаталог \vcpp32\Ex19a.**
2. **Напишите текст основного файла справки.** В Microsoft Word (или другом RTF-совместимом текстовом редакторе) наберите текст.



Проверьте, правильно ли вы применили режимы двойного подчеркивания и скрытого текста и правильно ли поставлен конец страницы.

Примечание Чтобы увидеть скрытый текст, надо активизировать режим просмотра скрытого текста в своем текстовом редакторе. Например, в Microsoft Word 2000 выберите в меню Tools (Сервис) команду Options (Параметры) и на вкладке View (Вид) и установите флажок All (Все) в секции Formatting Marks (Знаки форматирования).

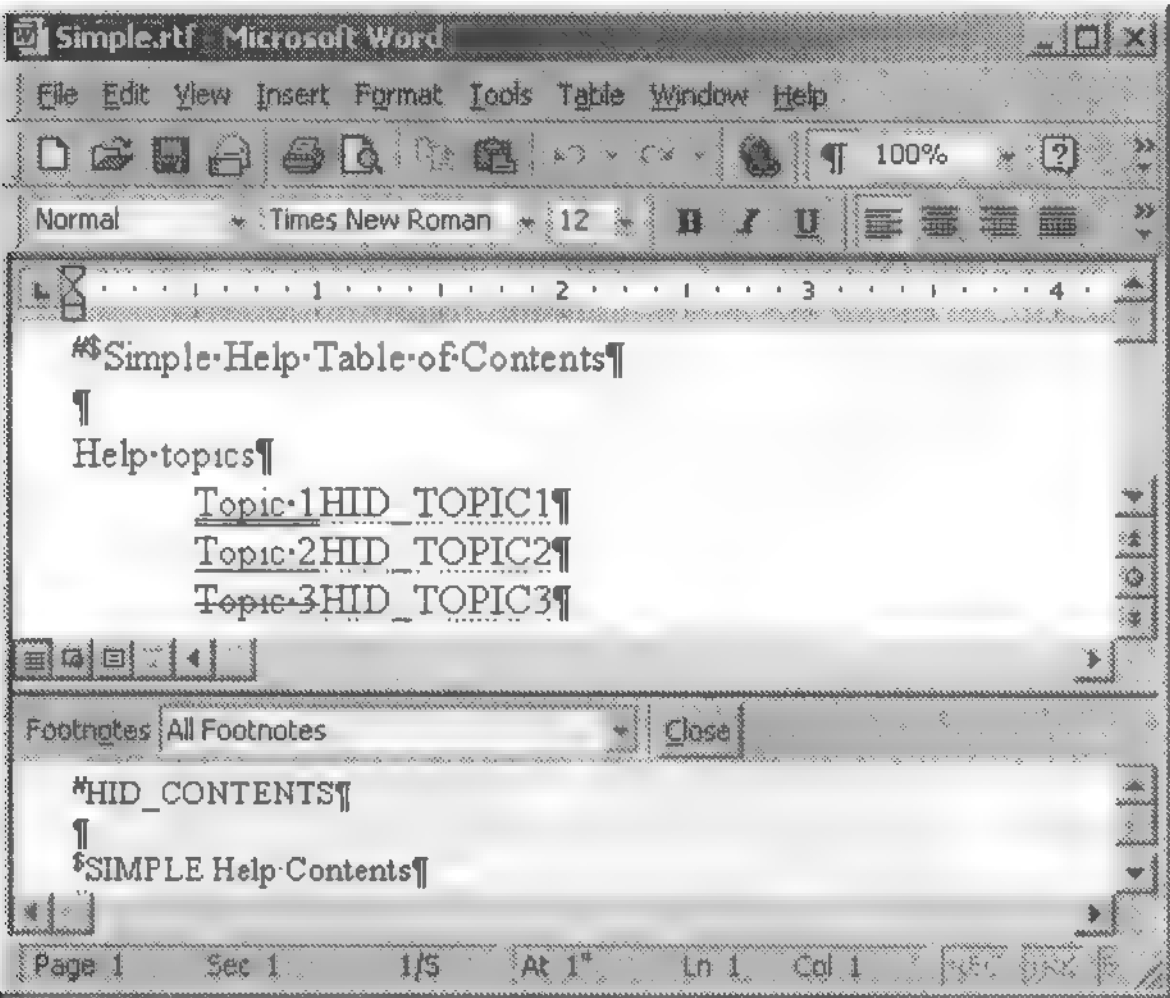
3. **Вставьте сноски для оглавления.** *Оглавление* (Table of Contents) — первая страница нашей справочной системы. Откройте в текстовом редакторе секцию сносок и вставьте перед заголовком тематического раздела следующие *сноски* (footnotes) с такими метками:

Метка	Текст сноски	Описание
#	HID_CONTENTS	Идентификатор контекста справки
$	SIMPLE Help Contents	Заголовок тематического раздела

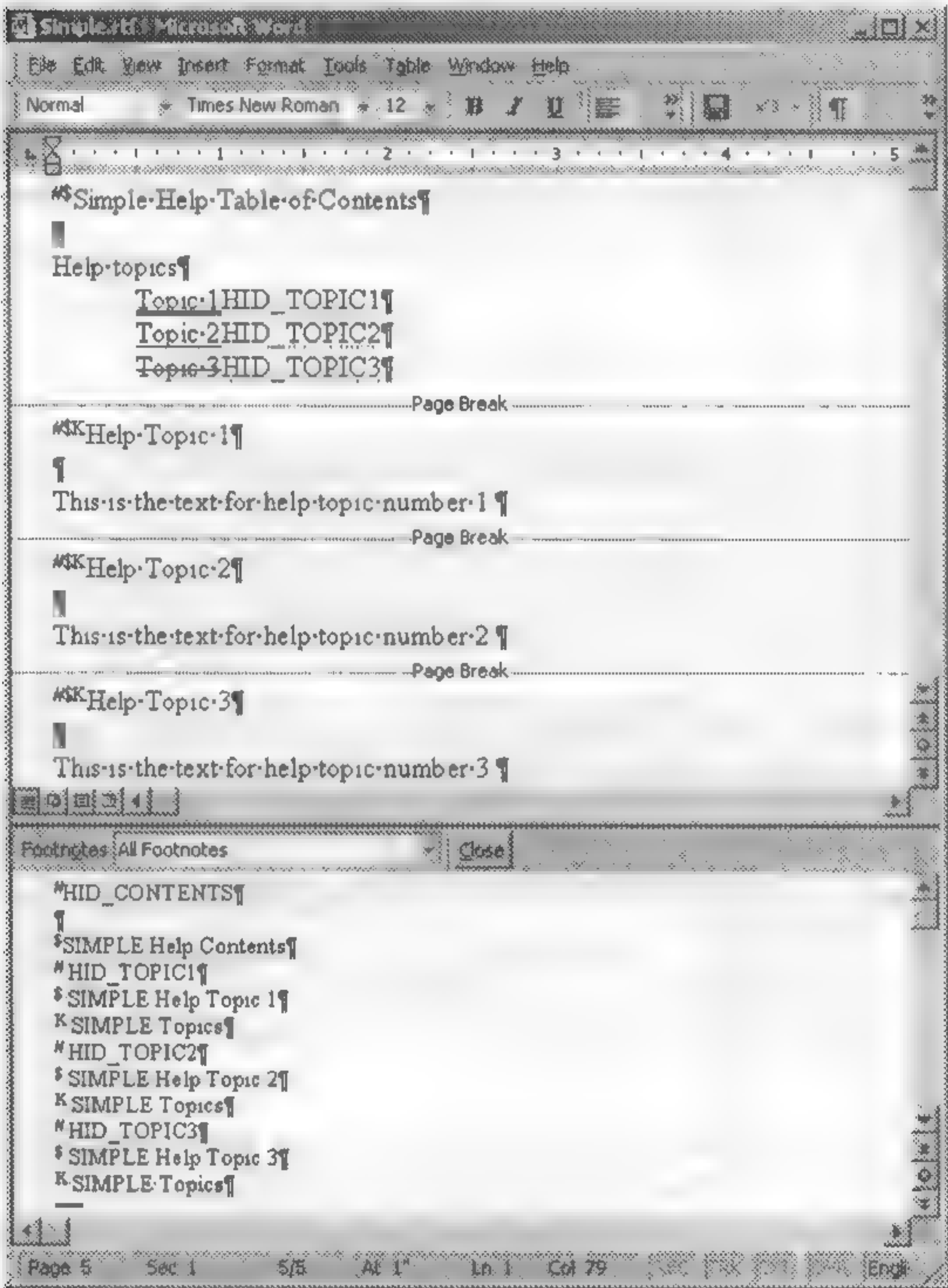
После этого документ должен выглядеть, как показано на рисунке ниже.

4. **Вставьте сноски для страницы Topic 1.** Topic 1 — вторая страница создаваемой системы. Вставьте следующие сноски с метками:

Метка	Текст сноски	Описание
#	HID_TOPIC1	Идентификатор контекста справки
$	SIMPLE Help Topic 1	Заголовок тематического раздела
K	SIMPLE Topics	Ключевые слова



5. **Дважды продублируйте страницу Topic 1.** Скопируйте раздел Topic 1, включая маркер конца страницы, в буфер обмена и вставьте в документ две его копии. Вместе с текстом скопируются и сноски. В первой копии замените все цифры *1* на *2*, во второй — на *3*. Не забудьте изменить и сноски. В Word бывает трудно сразу определить, к чему относится каждая сноска, так что будьте внимательны. Документ, включая сноски, после этой операции должен выглядеть так:

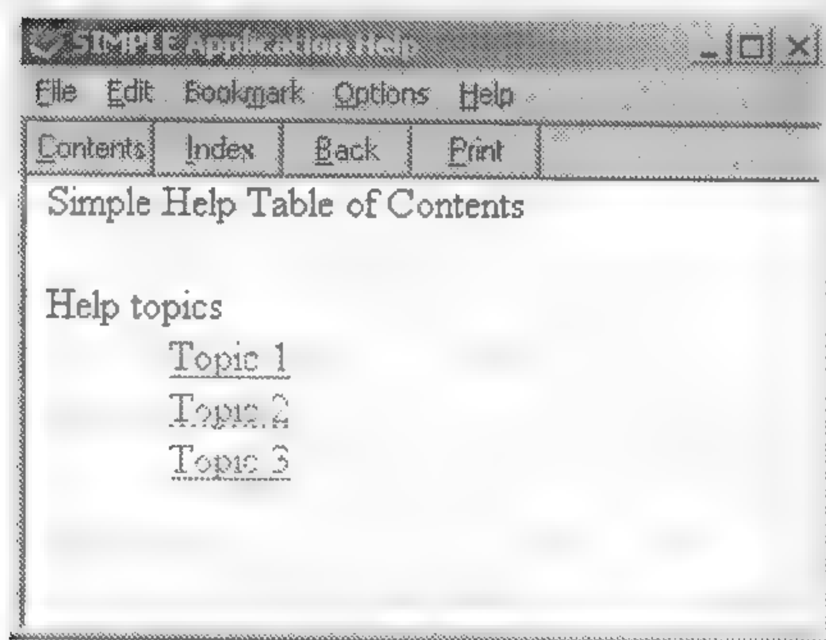


6. **Сохраните документ в файле \vcpp32\Ex19a\Simple.rtf.** В качестве типа файла выберите Rich Text Format.
7. **Подготовьте файл проекта справочной системы.** В Visual C++ .NET или другом текстовом редакторе создайте файл \vcpp32\Ex19a\Simple.hpj с текстом:

```
[OPTIONS]
CONTENTS=HID_CONTENTS
TITLE=SIMPLE Application Help
COMPRESS=true
WARNING=2

[FILES]
simple.rtf
```

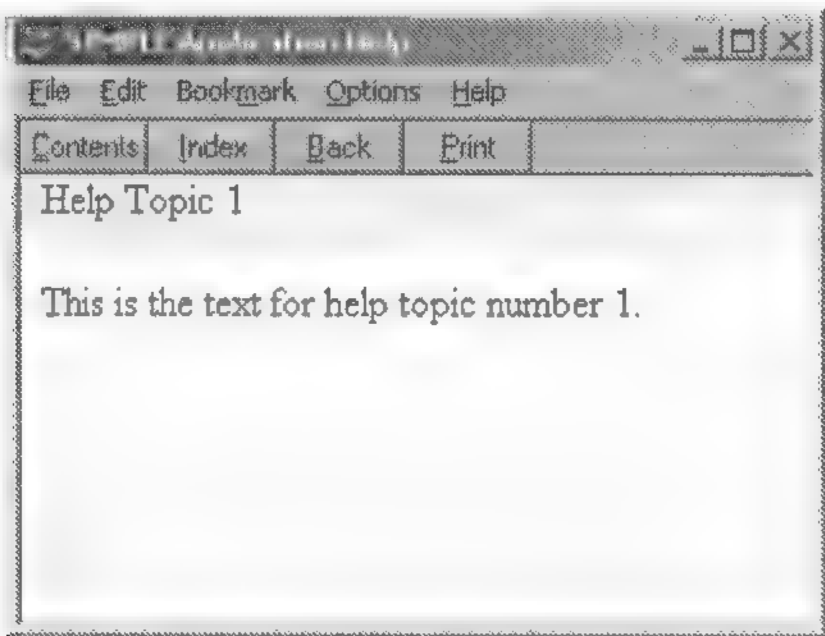
   Файл определяет идентификатор контекста справки для оглавления и имя RTF-файла с текстом справки. Сохраните файл в текстовом формате (ASCII).
8. **Соберите справочный файл.** Запустите в Windows утилиту Microsoft Help Workshop (HCRTF.exe) (она обычно хранится в каталоге Program Files\Microsoft Visual Studio\Common\Tools). Откройте файл \vcpp32\Ex19a\Simple.hpj и щелкните команду Compile из меню File. Запустится Windows Help Compiler с файлом проекта Simple.hpj. Компилятор сформирует справочный файл Simple.hlp в том же каталоге.
9. **Запустите WinHelp с новым справочным файлом.** В Windows Explorer дважды щелкните файл \vcpp32\Ex19a\Simple.hlp. Страница оглавления должна выглядеть так:

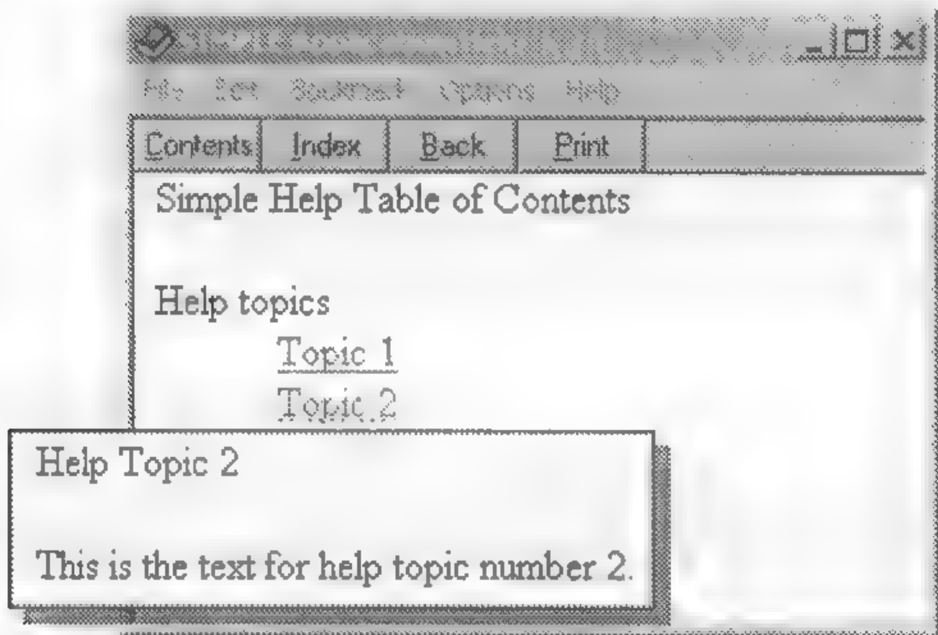


   Теперь переместите указатель мыши на строку Topic 1 и обратите внимание, как меняется его форма: из стрелки он превращается в кисть руки с «указующим перстом». Щелкните левую кнопку — откроется страница Topic 1 (см. первый рисунок на след. странице).

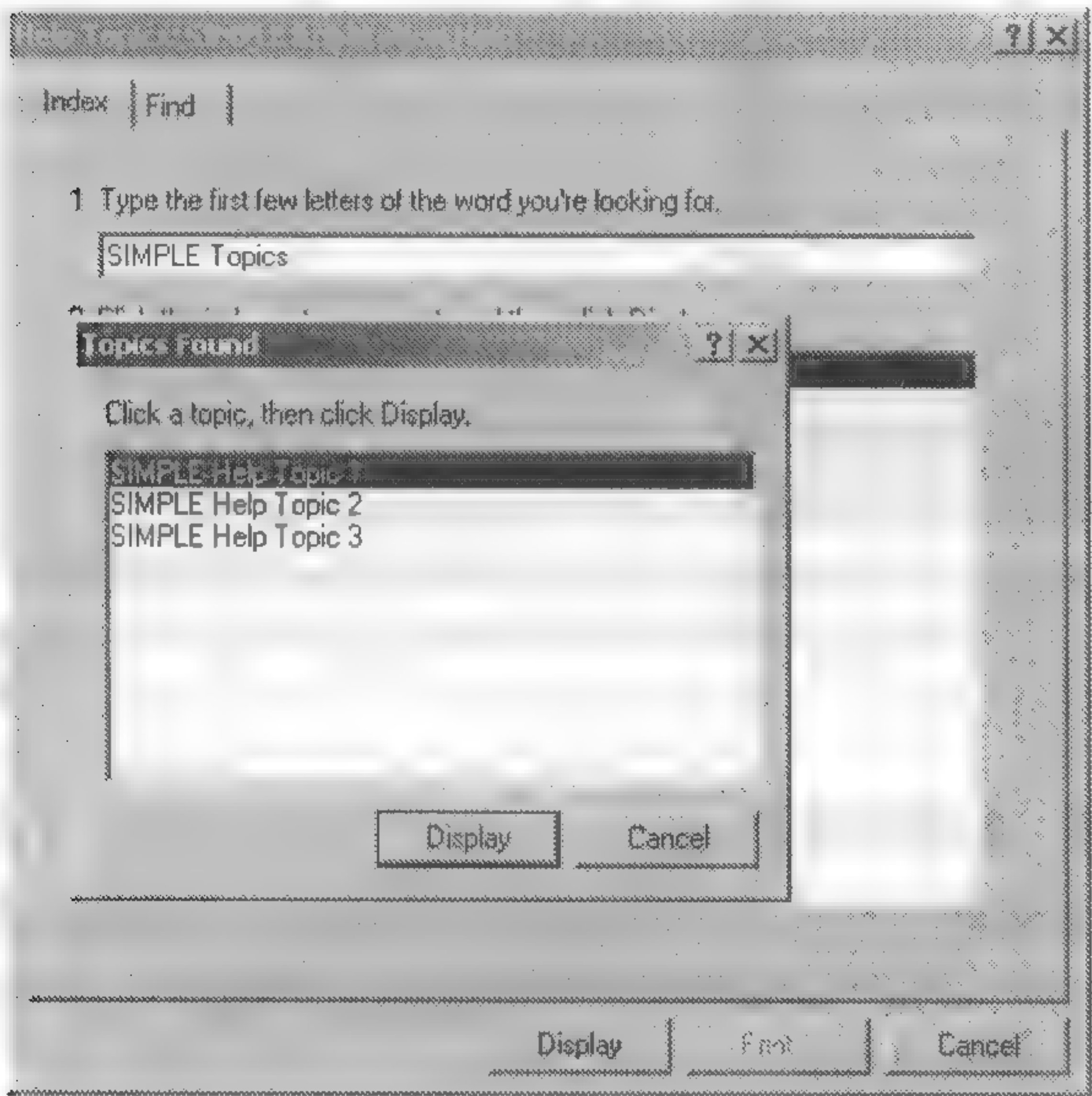
   Текст *HID_TOPIC1* в оглавлении связан с соответствующим идентификатором контекста (сноска #) на тематической странице. Эту связь называют также *переходом* (jump).

   Щелчок Topic 2 вызывает на экран всплывающее окно (см. второй рисунок на след. странице).





10. **Щелкните кнопку Contents в WinHelp.** Должна открыться страница оглавления (см. п. 9). Программе WinHelp известен идентификатор этого экрана, поскольку вы определили его в HPJ-файле.
11. **Щелкните кнопку Index в WinHelp.** WinHelp откроет диалоговое окно Index со списком ключевых слов в данном справочном файле. В нашем файле (Simple.hlp) во всех тематических разделах (кроме оглавления) только одно ключевое словосочетание — SIMPLE Topics (сноски K). Дважды щелкнув его, вы увидите связанные с ним тематические разделы (сноски $).



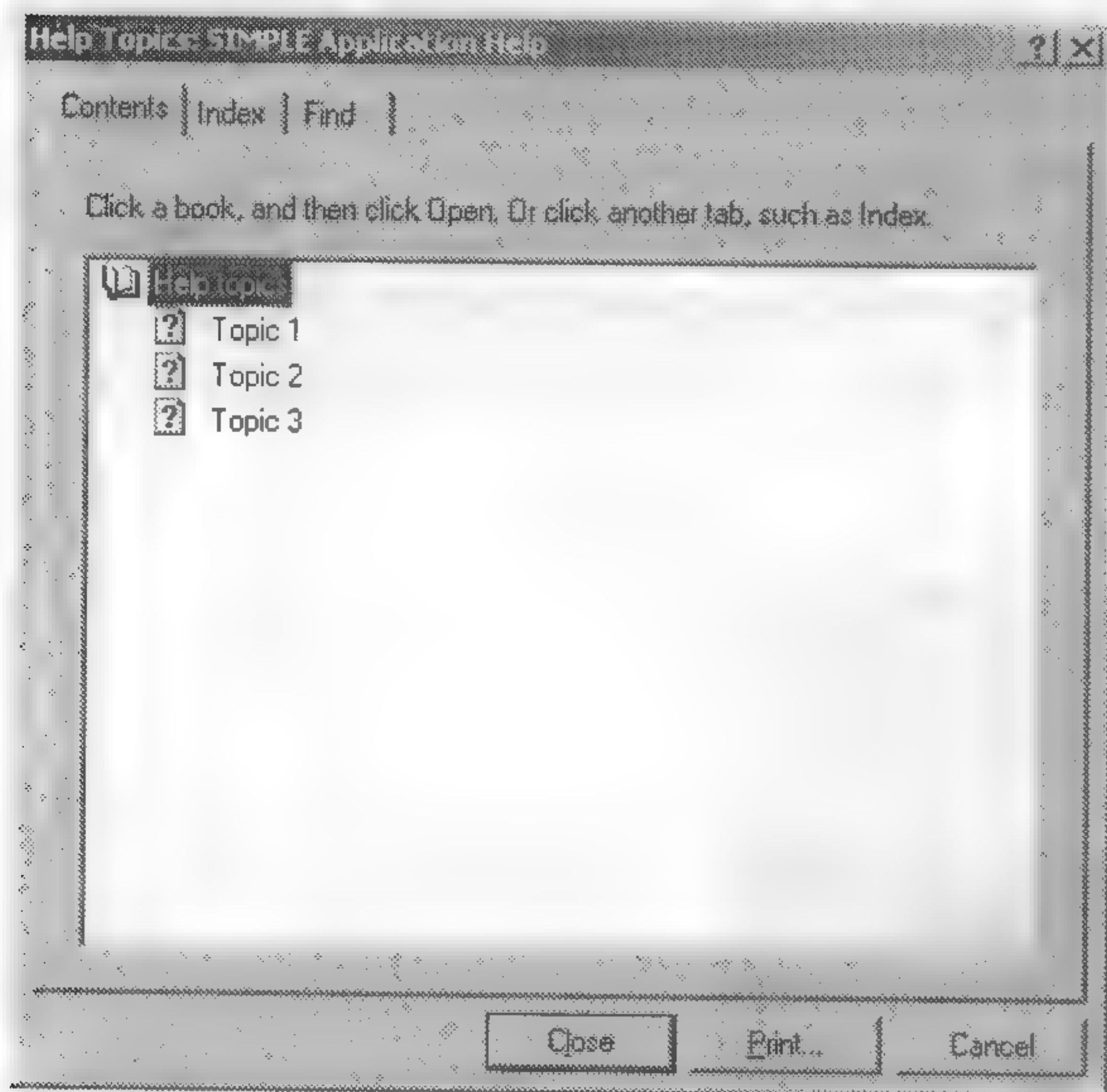
То, что мы получили, называется двухуровневой системой поиска справочной информации. Пользователь может ввести первые несколько букв ключевого слова и выбрать тему из списка. Чем тщательнее вы подберете ключевые слова и названия тематических разделов, тем эффективнее будет ваша справочная система.

Совершенствование оглавления

Оглавление, которое мы только что подготовили, сегодня считается старомодным. Последняя версия WinHelp для Win32 позволяет создавать современное оглавление с древовидной структурой. Все, что для этого нужно, — текстовый файл с расширением CNT. Добавьте в каталог \vcpp32\Ex19a новый файл Simple.cnt с таким текстом:

```
:Base simple.hlp
1 Help topics
2 Topic 1=HID_TOPIC1
2 Topic 2=HID_TOPIC2
2 Topic 3=HID_TOPIC3
```

Идентификаторы контекста соответствуют содержимому справочного файла. Скомпилировав Simple.cnt, при следующем запуске WinHelp с файлом Simple.hlp вы увидите новое оглавление.



CNT-файлы можно редактировать, используя HCRTF. CNT-файл не зависит от HPJ- и RTF-файлов. Но не забудьте, обновив свои RTF-файлы, внести соответствующие изменения и в CNT-файл.

Каркас приложений и WinHelp

Вы видели, что WinHelp может работать как автономная программа. В то же время с ней может взаимодействовать каркас приложений, обеспечивая тем самым

вывод контекстно-зависимой справки в ваших программах. Вот как вкратце это происходит.

1. При запуске MFC Application Wizard установите флажок Context-sensitive Help, а переключатель — в положение WinHelp (а не HTML Text).
2. MFC Application Wizard формирует в меню Help вашего приложения элемент Help Topics и создает один или несколько RTF-файлов, HPJ-файл и командный файл, запускающий компилятор справочной системы Help Compiler.
3. Затем MFC Application Wizard определяет F1 как «быструю клавишу» и связывает ее и элемент Help Topics меню Help с функциями-членами объекта основного окна-рамки.
4. Когда пользователь нажимает клавишу F1 или выбирает команду меню Help Topics, программа вызывает WinHelp, передавая ей идентификатор контекста, определяющий, какой именно раздел справки показать на экране.

Теперь разберемся, как вызвать WinHelp из программы и как последняя генерирует идентификаторы контекста для WinHelp.

Вызов WinHelp

Функция-член *CWinApp::WinHelp* активизирует WinHelp из приложения. Заглянув в интерактивную документацию, вы увидите длинный перечень операций, контролируемых необязательным (вторым) параметром функции *WinHelp*. Мы не станем его использовать и будем считать, что у *WinHelp* есть только один параметр — *dwData* типа *unsigned long int*. Он задает справочные темы. Пусть у нас есть справочный файл SIMPLE, а в нашей программе есть оператор:

```
AfxGetApp()->WinHelp(HID_TOPIC1);
```

После его исполнения (в ответ на нажатие клавиши F1 или другое событие) появляется страница Topic 1 справочной системы — как если бы пользователь выбрал этот раздел в оглавлении.

«Постойте-ка, — заметите вы, — а откуда WinHelp знает, какой справочный файл взять?» Дело в том, что имя справочного файла соответствует имени программы. Например у программы с исполняемым файлом Simple.exe справочный файл называется Simple.hlp.

Примечание Можно заставить WinHelp использовать другой справочный файл, записав нужное значение в переменную-член *m_pszHelpFilePath* класса *CWinApp*.

«А как WinHelp связывает константу *HID_TOPIC1* с идентификатором контекста справочного файла?» — спросите вы. Файл проекта справки должен содержать раздел MAP, в котором идентификаторы контекста увязаны с конкретными числами. Если в файле resource.h приложения *HID_TOPIC1* определен как *101*, то раздел MAP файла simple.hpj выглядит так:

```
[MAP]
HID_TOPIC1    101
```

Имена констант (заданных в программе операторами *#define*) не обязательно должны совпадать с идентификаторами контекста — главное, чтобы совпадали их численные значения. И все же соответствие имен считается в программировании хорошим тоном.

Поиск строк

В программе, оперирующей с текстом, может понадобиться поиск справки по ключевому слову, а не по числовому идентификатору контекста. В этом случае используйте необязательный параметр *HELP_KEY* или *HELP_PARTIALKEY* функции *WinHelp*:

```
CString string("find this string"); // ищем эту строку
AfxGetApp()->WinHelp((DWORD) (LPCSTR) string, HELP_KEY);
```

Двойное приведение типа для *string* необходимо потому, что первый параметр *WinHelp* — многоцелевой; его смысл зависит от значения второго параметра.

Вызов WinHelp из меню программы

MFC Application Wizard генерирует в меню Help элемент Help Topics, сопоставляя его функции *CWnd::OnHelpFinder* в основном окне-рамке, которая обращается к WinHelp так:

```
AfxGetApp()->WinHelp(0L, HELP_FINDER);
```

При этом вызове WinHelp отображает страницу оглавления, позволяя пользователю перемещаться по файлу справки при помощи переходов и поиска текста.

Если вы хотите получить «старомодное» оглавление, напишите:

```
AfxGetApp()->WinHelp(0L, HELP_INDEX);
```

А если вам нужна справка по работе со справкой, вызовите:

```
AfxGetApp()->WinHelp(0L, HELP_HELPONHELP);
```

HELPONHELP — стандартный идентификатор, требующий отобразить справку по работе с самой справочной системой, однако он работает только при наличии файла Winhlp32.hlp.

Синонимы контекстной справки

Раздел ALIAS в HPJ-файле позволяет отождествлять разные идентификаторы контекста. Допустим, в вашем HPJ-файле есть операторы:

```
[ALIAS]
HID_TOPIC1 = HID_GETTING_STARTED

[MAP]
HID_TOPIC1   101
```

Тогда в RTF-файлах идентификаторы *HID_TOPIC1* и *HID_GETTING_STARTED* взаимозаменяемы и оба связаны с контекстом справки *101*.

Определение контекста справки

Теперь вы уже знаете, как создать в MFC-программе простую контекстно-зависимую справочную систему. Вы определяете F1 как «быструю клавишу» (в MFC-библиотеке это стандартная клавиша для вызова контекстной справки) и пишете обработчик команды, связывающий контекст справки с параметром функции *WinHelp*. Можно было бы придумать свой способ сопоставить состояние программы контекстному идентификатору, но отчего не воспользоваться преимуществами системы, уже встроенной в каркас приложений?

Каркас приложений определяет контекст справки на основе идентификатора активного элемента программы. К последним относятся команды меню, окна-рамки, диалоговые и информационные окна, а также элементы управления. Элемент меню можно идентифицировать, скажем, как *ID_EDIT_CLEARALL*, а у основного окна-рамки обычно идентификатор *IDR_MAINFRAME*. Можно было бы ожидать, что все эти идентификаторы прямо связаны с контекстами справки. А если идентификаторы команды и окна-рамки имеют одно и то же численное значение? Очевидно, таких накладок лучше как-то избежать.

В каркасе приложений эта проблема решается путем определения нового набора *#define*-констант справки производных от идентификаторов программных элементов. Эти константы представляют собой сумму идентификаторов программных элементов и определенных базовых значений.

Элемент программы	Префикс идентификатора элемента	Префикс элемента контекста справки	Базовое значение (шестнадцатеричное)
Элемент меню или кнопка панели инструментов	*ID_, IDM_*	*HID_, HIDM_*	10000
Окно-рамка или диалоговое окно	*IDR_, IDD_*	*HIDR_, HIDD_*	20000
Окно сообщений об ошибках	*IDP_*	*HIDP_*	30000
Неклиентская область		*H..*	40000
Элемент управления	*IDW_*	*HIDW_*	50000
Сообщения об ошибках диспетчеризации			60000

HID_EDIT_CLEARALL (0x1E121) соответствует ID_EDIT_CLEARALL (0xE121), а HIDR_MAINFRAME (0x20080) — IDR_MAINFRAME (0x80).

Вызов справки клавишей F1

В MFC-программу встраиваются еще два способа доступа к контекстно-зависимой справке; они доступны при установке в MFC Application Wizard флажка Context-sensitive Help. Первый способ — стандартный вызов справки при нажатии клавиши F1. Пользователь нажимает ее, а программа, определив контекст справки, вызывает WinHelp. Этот способ позволяет задать вывод справки об элементе меню, выбранном с клавиатуры, или о текущем окне (рамке, представлении, диалоговом или информационном окне).

Вызов справки сочетанием клавиш Shift+F1

Второй способ вывода контекстно-зависимой справки предоставляет больше возможностей, чем первый. В нем программа различает такие контексты справки:

- элемент меню, на который указывает указатель мыши;
- кнопку на панели инструментов;
- окно-рамку;
- окно представления;
- конкретный графический элемент в окне представления;
- строку состояния;
- различные неклиентские элементы такие, как команды системного меню.

Примечание Справка по нажатию Shift+F1 не работает с модальными диалоговыми и информационными окнами.

Пользователь активизирует такую справку, нажимая сочетание клавиш Shift+Fl или щелкая кнопки Context Help на панели инструментов. В любом случае указатель мыши меняет свою форму на значок со знаком вопроса. При следующем щелчке появляется справка контекста, определяемого по позиции курсора.

Справка в информационном окне: функция *AfxMessageBox*

Глобальная функция *AfxMessageBox* выводит сообщения об ошибках, формируемые каркасом приложений. Она аналогична функции-члену *CWnd::MessageBox*, но как дополнительный параметр получает идентификатор контекста справки. Каркас приложений сопоставляет его идентификатору контекста WinHelp и вызывает WinHelp, когда пользователь нажимает клавишу F1. Если вы хотите указать упомянутый параметр *AfxMessageBox*, имейте в виду, что соответствующие идентификаторы должны начинаться *с IDP_*, а контексты справки в RTF-файле — с *HIDP_*.

Существует два варианта *AfxMessageBox*. В первом строка подсказки определяется параметром — указателем на символьный массив. Во втором (с ним программы работают эффективнее) параметр — идентификатор подсказки определяет строковый ресурс. У обоих вариантов *AfxMessageBox* есть параметр «стиль», заставляющий показывать в информационном окне восклицательный или вопросительный знак либо иной графический символ.

Стандартные разделы справочной системы

При установленном флажке Context-sensitive Help мастер MFC Application Wizard формирует справочные разделы, связанные со стандартными элементами MFC-программы:

- стандартные команды меню и кнопки на панели инструментов (File, Edit и т. п.);
- неклиентские элементы окна (кнопка развертывания окна, строка заголовка и т. д.);
- строка состояния;
- окна сообщений об ошибках.

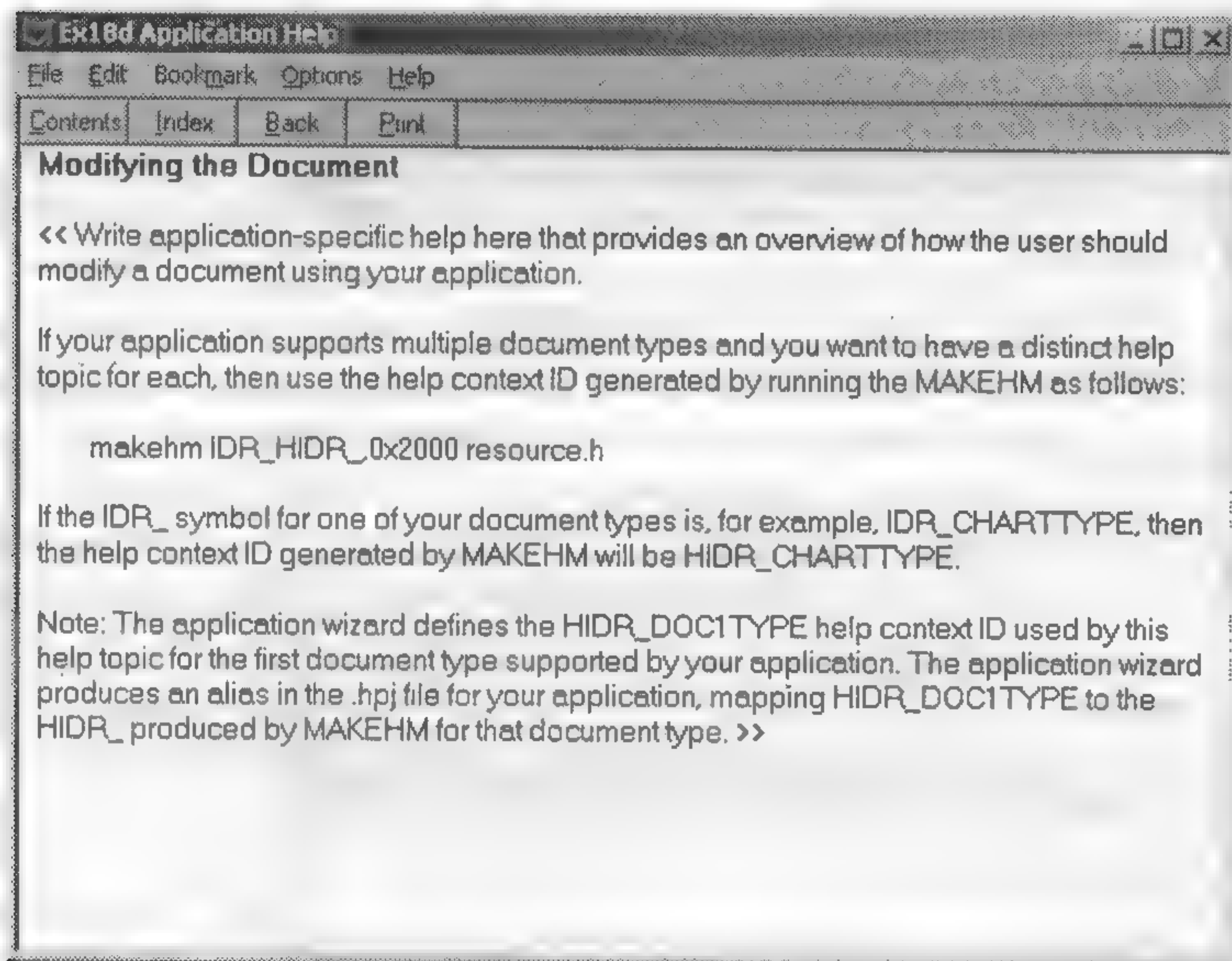
Справочные разделы по этим элементам содержатся в файлах AfxCore.rtf и AfxPrint.rtf, размещаемых вместе с соответствующими файлами растровых изображений в подкаталоге HLP проекта. Ваша работа — адаптировать эти файлы в соответствии с требованиями к вашей программе.

Примечание MFC Application Wizard генерирует AfxPrint.rtf, только если установлен флажок Printing and print preview.

Пример создания справочной системы без программирования

Работая над программой-примером Ex18d в главе 18, вы устанавливали флажок Context-sensitive Help в MFC Application Wizard. Вернемся к тому примеру и исследуем «справочные» возможности, встроенные в каркас приложений. Вы увидите, насколько легко связать справочные разделы с идентификаторами команд меню и ресурсов окна-рамки. Мы отредактируем RTF-, а не CPP-файлы.

1. **Проверьте правильность построения справочного файла.** Если вы уже собрали проект Ex18d, то велика вероятность, что справочный файл уже создан в ходе процесса сборки. Проверьте это, запустив приложение и нажав клавишу F1. Вы должны увидеть стандартный экран Application Help с заголовком «Modifying the Document»:



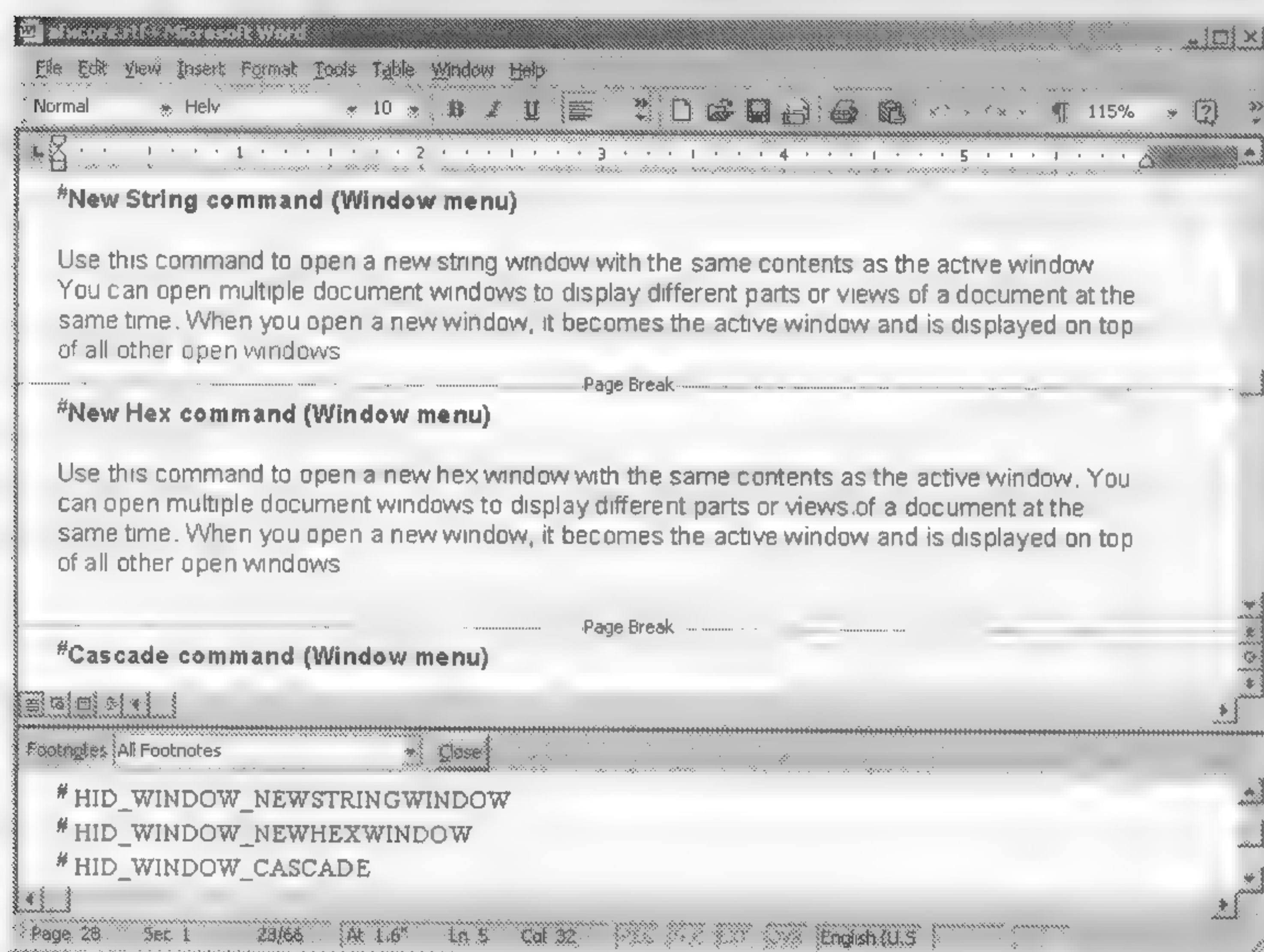
Если вы не увидели такого окна, значит, справочный файл создан некорректно. Соберите приложение повторно, запустите Ex18d и вновь нажмите F1.

2. **Проверьте стандартный справочный файл.** Проведите эксперименты.
   - Закройте диалоговое окно справки и нажмите сочетание клавиш Alt+F, а затем F1. Должно открыться окно со справочным разделом о команде New из меню

File. Удерживая указатель мыши на команде New меню File и нажав клавишу F1, вы должны увидеть ту же страницу справки.

- ☐ Закройте окно справки, нажмите кнопку Context Help на панели инструментов и выберите из меню File команду Save. Открылся ли соответствующий раздел справки?
- ☐ Снова нажмите кнопку Context Help на панели инструментов и щелкните строку заголовка окна-рамки. Вы должны получить справку по заголовкам окон в Windows.
- ☐ Закройте все дочерние окна и нажмите F1. Вы должны увидеть основную страницу предметного указателя, которая представляет собой «старомодное» оглавление.

3. **Измените заголовок приложения.** В файле AfxCore.rtf (в каталоге \vcpp32\Ex18d\hlp) есть несколько строк <<*YourApp*>>. Замените их повсюду на *Ex18d*.
4. **Отредактируйте раздел Modifying The Document справочной системы.** Этот текст хранится в файле AfxCore.rtf (каталог \vcpp32\Ex18d\hlp). Найдите раздел *Modifying The Document* и замените его текст своим, подходящим для вашего приложения. Идентификатор контекста справки у этого раздела — *HIDR_DOC1TYPE*. Сгенерированный файл ex20d.hpj предоставляет синоним *HIDR_Ex18dTYPE*.
5. **Добавьте раздел для команды New String Window в меню Window.** Эта команда добавлена в Ex18d нами, и поэтому для нее нет справочного текста. Введите нужный текст в AfxCore.rtf:



Не забудьте о сноске #, связывающей раздел с идентификатором контекста *HID_WINDOW_NEW_STRING*, определенным в hlp\Ex18d.hm. Элемент меню New String Window, конечно же, имеет идентификатор команды *ID_WINDOW_NEWSTRINGWINDOW*.

6. **Соберите и протестируйте приложение.** Пересоберите приложение, чтобы синхронизировать файлы справки. Проверьте две новые ссылки в справке.

Обработка команд вызова справки

Итак, вы увидели, из каких компонентов состоит справочный файл и как действуют клавиша F1 и сочетание Shift+F1. Вы знаете, как идентификаторы программных элементов связаны с идентификаторами контекста справки. Однако внутренние механизмы обработки команд запроса справки каркасом приложений мы пока не обсуждали. А нужно ли это? Хм, представьте себе: вы хотите дать справку о каком-то окне представления (а не об окне-рамке) и собираетесь связать справочные разделы с теми или иными графическими элементами в данном окне. И это, и многое другое можно реализовать, только переопределив функции, отвечающие за обработку команд в меню Help.

Такая обработка зависит от того, как запрошена справка: по нажатию F1 или Shift+F1. Рассмотрим их обработку по отдельности.

Обработка клавиши F1

Клавишу F1 обычно обрабатывает команда, указанная в соответствующей записи таблицы быстрых клавиш, которую MFC Application Wizard вставляет в RC-файл. Клавиша F1 сопоставляется команде *ID_HELP*, а та — функции-члену *OnHelp* класса *CFrameWnd*.

Примечание В активном модальном диалоговом окне или в процессе выбора из меню клавиша F1 обрабатывается Windows-ловушкой, которая вызывает ту же функцию *OnHelp*. В противном случае быстрая клавиша F1 была бы заблокирована.

Функция *CFrameWnd::OnHelp* посылает MFC-сообщение *WM_COMMANDHELP* внутреннему окну на самом нижнем уровне вложения — это обычно окно представления. Если в классе «вид» нет обработчика этого сообщения или обработчик возвращает *FALSE*, каркас приложений направляет сообщение вверх следующему по иерархии окну (как правило, это дочернее окно-рамка MDI или основное окно-рамка). Если в ваших производных классах окон-рамок не определены обработчики *WM_COMMANDHELP*, сообщение обрабатывается в MFC-классе *CFrameWnd*. Он отображает справку для символа, который определил MFC Application Wizard для вашего приложения или типа документа.

Если вы определили обработчик *WM_COMMANDHELP* в производных классах, ваш обработчик должен вызывать *CWinApp::WinHelp* с корректным идентификатором контекста в качестве параметра.

Для любого приложения MFC Application Wizard добавляет к проекту символ *IDR_MAINFRAME*, а HM-файл определяет идентификатор контекста справки *HIDR_MAINFRAME*, который сопоставляется с *main_index* в HPJ-файле. Стандартный файл AfxCore.rtf ассоциирует основной предметный указатель с этим идентификатором контекста.

Например, для MDI-приложения SAMPLE мастер MFC Application Wizard внесет в проект символ *IDR_SAMPLETYPE*, а в HM-файле определит идентификатор контекста *HIDR_SAMPLETYPE*, сопоставляемый в HPJ-файле символу *HIDR_DOCITYPE*. Стандартный файл AfxCore.rtf ассоциирует с этим идентификатором контекста страницу «Modifying the Document».

Обработка сочетания клавиш Shift+F1

При нажатии сочетания клавиш Shift+F1 или щелчке кнопки Context Help на панели инструментов функция *CFrameWnd::OnContextHelp* получает сообщение, связанное с соответствующей командой меню. Когда пользователь вновь делает щелчок, поместив указатель на каком-нибудь элементе экрана, окну ниже по иерархии (из числа тех, где обнаружен щелчок) посылается сообщение *WM_HELPHITTEST*. С этого момента маршрутизация сообщения происходит так же, как сообщения *WM_COMMANDHELP* (см. предыдущий раздел).

Параметр *lParam* функции *OnHelpHitTest* содержит координаты мыши в единицах устройства относительно верхнего левого угла клиентской области окна. В старшем «слове» хранится координата *y*, а в младшем — *x*. Эти координаты позволяют указывать идентификатор контекста справки для конкретного элемента окна представления. Обработчик *OnHelpHitTest* должен возвращать правильный идентификатор контекста; вызовом каркаса приложения займется *WinHelp*.

Пример Ex19b: обработка команд вызова справки

Эта программа основана на примере Ex18d из главы 18 и представляет собой MDI-приложение с двумя окнами представления; в ее справочную систему добавлены разделы, относящиеся к каждому из этих окон. В каждом из двух классов «вид» есть обработчики сообщений *OnCommandHelp* (для F1) и *OnHelpHitTest* (для комбинации Shift+F1).

Требования к заголовочным файлам

Компилятор распознает особые идентификаторы для справки, только если присутствует оператор *#include*:

```
#include <afxpriv.h>
```

В Ex19b такой оператор имеется в файле StdAfx.h.

CStringView

Классу, отвечающему за строковое представление документа, в StringView.h нужны прототипы функций таблицы сообщений для справки как по F1, так и Shift+F1:

```
afx_msg LRESULT OnCommandHelp(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnHelpHitTest(WPARAM wParam, LPARAM lParam);
```

а в StringView.cpp — следующие записи таблицы сообщений:

```
ON_MESSAGE(WM_COMMANDHELP, OnCommandHelp)
ON_MESSAGE(WM_HELPHITTEST, OnHelpHitTest)
```

Функция-член *OnCommandHelp* в StringView.cpp обрабатывает запросы справки по F1. Она реагирует на сообщение, присланное из основного окна-рамки MDI, и отображает раздел справки для окна строкового представления:

```
LRESULT CStringView::OnCommandHelp(WPARAM wParam, LPARAM lParam)
{
   if (lParam == 0) {     // контекст еще не определен
```

```
        lParam = HID_BASE_RESOURCE + IDR_STRINGVIEW;
    }
    AfxGetApp()->WinHelp(lParam);
    return TRUE;
}
```

Функция-член *OnHelpHitTest* обрабатывает справку по Shift+F1:

```
LRESULT CStringView::OnHelpHitTest(WPARAM wParam, LPARAM lParam)
{
    return HID_BASE_RESOURCE + IDR_STRINGVIEW;
}
```

В более сложной программе может потребоваться, чтобы *OnHelpHitTest* устанавливала контекст справки по позиции курсора.

CHexView

Класс *CHexView* обрабатывает запросы справки так же, как и класс *CStringView*. В HexView.h обязателен код:

```
afx_msg LRESULT OnCommandHelp(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnHelpHitTest(WPARAM wParam, LPARAM lParam);
```

Строки таблицы сообщений в HexView.cpp выглядят так:

```
ON_MESSAGE(WM_COMMANDHELP, OnCommandHelp)
ON_MESSAGE(WM_HELPHITTEST, OnHelpHitTest)
```

И, наконец, код реализации в HexView.cpp:

```
LRESULT CHexView::OnCommandHelp(WPARAM wParam, LPARAM lParam)
{
    if (lParam == 0) {     // контекст еще не определен
        lParam = HID_BASE_RESOURCE + IDR_HEXVIEW;
    }
    AfxGetApp()->WinHelp(lParam);
    return TRUE;
}

LRESULT CHexView::OnHelpHitTest(WPARAM wParam, LPARAM lParam)
{
    return HID_BASE_RESOURCE + IDR_HEXVIEW;
}
```

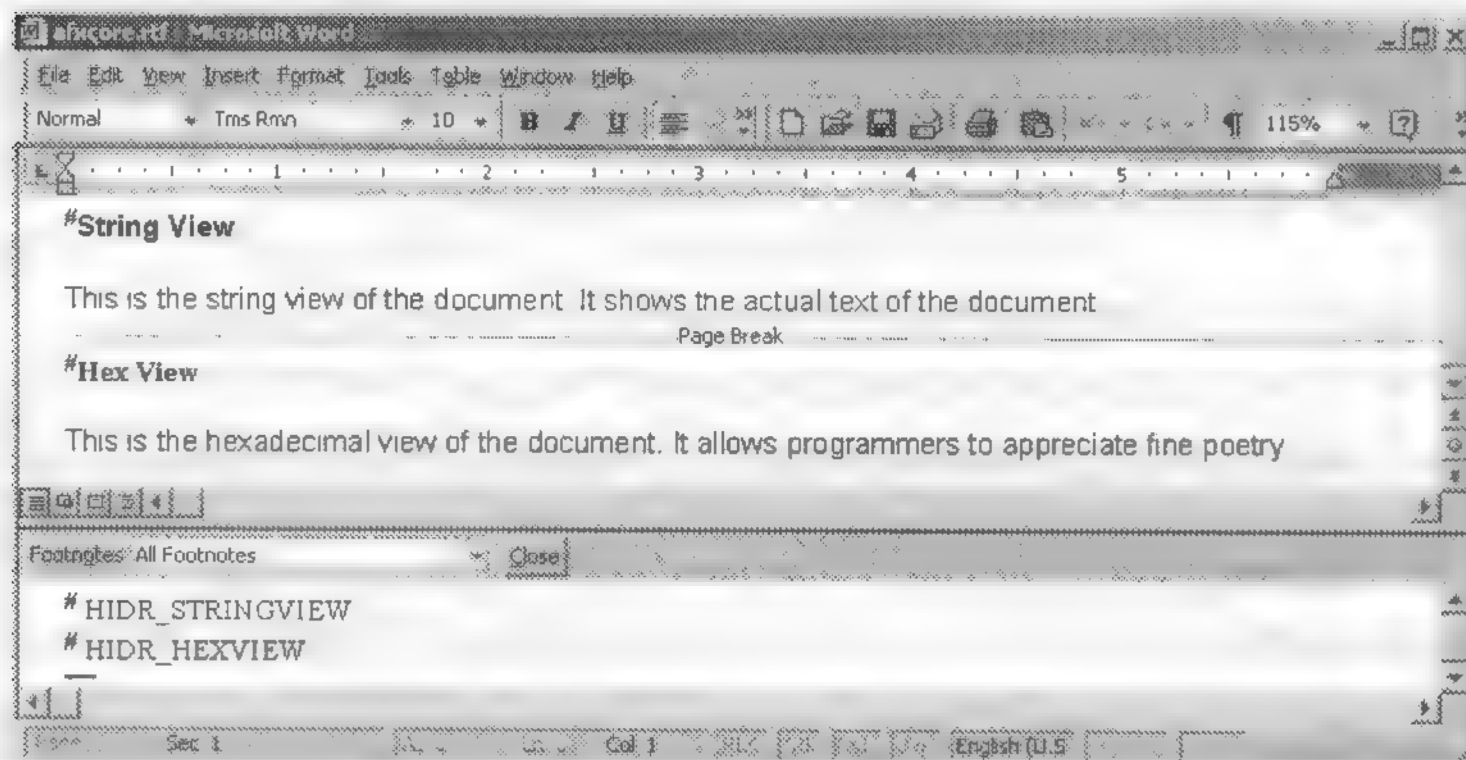
Требования к ресурсам

В файл ресурсов добавлены два символа:

Символ	Значение	Идентификатор контекста справки	Значение
IDR_STRINGVIEW	101	*HIDR_STRINGVIEW*	0x20065
IDR_HEXVIEW	102	*HIDR_HEXVIEW*	0x20066

Требования к справочному файлу

В файл AfxCore.rtf добавлены два справочных раздела — с идентификаторами *HIDR_STRINGVIEW* и *HIDR_HEXVIEW*.



Сгенерированный в подкаталоге HLP проекта файл Ex19b.hm должен выглядеть так:

```
// Commands (ID_* and IDM_*)
HID_WINDOW_NEWHEXWINDOW          0x10082
HID_WINDOW_NEWSTRINGWINDOW       0x10083

// Prompts (IDP_*)
HIDP_OLE_INIT_FAILED             0x30064

// Resources (IDR_*)
HIDR_MANIFEST                    0x20001
HIDR_MAINFRAME                   0x20080
HIDR_Ex19bTYPE                   0x20081
HIDR_STRINGVIEW                  0x20065
HIDR_HEXVIEW                     0x20066

// Dialogs (IDD_*)
HIDD_ABOUTBOX                    0x20064

// Frame Controls (IDW_*)
```

Тестирование приложения Ex19b

Откройте дочерние окна со строковым и шестнадцатеричным представлением документа. Проверьте, как работает справочная система при нажатии клавиши F1 и сочетания клавиш Shift+F1.

MFC и HTML Help

В MFC-приложениях обращение к HTML Help выполняется так же, как и к файлу WinHelp. Приложение предоставляет контекстно-зависимой справочной системе идентификаторы справки, и она отображает окно с соответствующим разделом

справки. Однако файлы HTML Help скроены на иной лад — они скомпилированы из набора HTML-страниц, а не из одного RTF-файла.

В таблице перечислены файлы, создаваемые мастером MFC Application Wizard при выборе формата HTML Help для справочной системы.

Файл	Описание
HTMLDefines.h	Содержит идентификаторы контекста для всего проекта.
Документы справки HTML Help	HTML-файлы с текстом справки — обычно один на тематический раздел.
<*имя_проекта*>.hhc	Файл для компилятора HTML Help. Содержит команды о том, как компилировать содержимое справки HTML Help.
<*имя_проекта*>.hhp	Файл HTML Help для компилятора. Содержит директивы для компиляции содержимого справки HTML Help.
Main_index.htm	HTML-файл верхнего уровня. Здесь добавляются тематические разделы справочной системы.

Давайте поближе познакомимся с тем, как справочная система на базе HTML Help подключается к MFC-приложению.

Пример Ex19c: HTML Help

Ex19c также основан на примере Ex18d. Это MDI-приложение с двумя представлениями одного документа и поддержкой HTML Help. Пример создан с помощью MFC Application Wizard и переключателем на странице Advanced Features в положении HTML Help Format.

В файле *hid_window_newstringwindow.htm* вы увидите измененный текст справки, относящийся к команде меню New String Window. Этот текст отображается при выборе контекстно-зависимой справки для команды New String Window. Обратите внимание и на новый файл *hid_window_newhex.htm* (он не создан мастером MFC Application Wizard) — это страница справки для команды New Hex Window.

Если вы чувствуете в себе силы, можете изменять справочные файлы в Notepad. Файл содержит HTML-тэги, поэтому при достаточном умении редактирование не составляет труда. Однако лучше все-таки открыть HTML-файл в Visual Studio .NET — она «понимает» HTML-файлы и заметно облегчает их редактирование.

«А как вообще создать новые тематические разделы справки?» — спросите вы. Проще всего взять существующий HTML-файл, переименовывать его и заполнить нужным текстом. Затем новую HTML-страницу сопоставляют идентификатору команды (об этом ниже).

В Visual Studio .NET новые идентификаторы контекста справки добавляются при создании в программе новых команд меню и последующей ее компиляции. Где-то в начале файла HTMLDefines.h вы найдете строки, определяющие контексты справки для команд меню New String Window и New Hex Window:

```
#define HID_WINDOW_NEWSTRINGWINDOW          0x10082
#define HID_WINDOW_NEWHEXWINDOW             0x10083
```

В дополнение к стандартной справке для команды меню, созданной MFC Application Wizard, в Ex19c есть новый раздел для команды New Hex Window. Идентификатор контекста справки заботливо предоставлен Visual Studio .NET в момент

создания команды меню. Теперь нужно его связать с файлом *hid_window_newhexwindow.htm*. Строка в Ex19c.hhp файле сопоставляет идентификатор контекста справки с файлом:

```
hid_window_newhexwindow          = hid_window_newhexwindow.htm
```

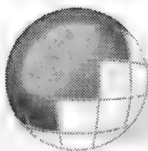
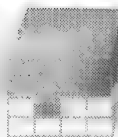
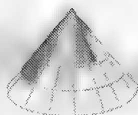
Наконец, файл Ex19c.hhp содержит ссылку на новый HTML-файл в разделе [FILES]:

```
[FILES]
afx_hidd_color.htm
afx_hidd_fileopen.htm
afx_hidd_filesave.htm
:
hid_window_newstringwindow.htm
hid_window_newhexwindow.htm
hid_window_split.htm
:
```

После создания связи HTML-файлов с идентификаторами справки по командам меню ничего делать не нужно — справочная система заработает автоматически. Об этом позаботится сама MFC. Чтобы убедиться лично, запустите программу Ex19c, последовательно выбирайте разные команды меню и нажимайте F1. Вы увидите, как по нажатию F1 открываются «правильные» страницы справки.

ГЛАВА

20

Динамически подключаемые библиотеки

Динамически подключаемые библиотеки (Dynamic-link libraries, DLL) остаются в основе компонентной модели Microsoft Windows даже в преддверии царства CLR-среды Microsoft .NET. Сама Windows состоит из DLL-библиотек, представляющих собой бинарные модули. Бинарная модульность отличается от модульности исходного кода в C++. Вместо гигантских EXE-файлов, которые пришлось бы перестраивать и тестировать при любом, даже самом незначительном изменении кода, гораздо эффективнее создавать компактные DLL-модули и тестировать их по отдельности. Если, например, выделить в DLL какой-нибудь класс C++, то после компиляции и компоновки объем модуля вряд ли превысит 12 кб. В период выполнения клиентские программы смогут очень быстро загружать и подключать ваши DLL.

Сегодня писать DLL стало гораздо легче. В Win32 модель программирования резко упростилась, да и со стороны MFC Application Wizard и MFC поддержка DLL расширена и улучшена. В этой главе вы научитесь писать на C++ DLL-модули и клиентские программы, способные их использовать. Вы увидите, как Win32 проецирует DLL на адресные пространства процессов, и узнаете о различиях между обычными DLL MFC-библиотеки (regular DLL) и DLL-расширениями (extension DLL). И, конечно же, мы рассмотрим простые примеры DLL всех типов и даже более сложный пример DLL, в которой реализован *пользовательский элемент управления* (custom control).

Основы DLL

Прежде чем обсуждать поддержку DLL каркасом приложений, надо разобраться, как Win32 интегрирует эти модули в процессы. Может, стоит даже вернуться к главе 10 и еще раз прочесть о процессах и о виртуальной памяти. Помните, что

процесс — это выполняемый экземпляр программы, а программа запускается из EXE-файла на диске.

В общем, DLL-модуль — это файл на диске (обычно с расширением DLL), который состоит из глобальных данных, скомпилированных функций и ресурсов и становится частью вашего процесса. Он компилируется для загрузки по определенному базовому адресу; при отсутствии конфликтов с другими DLL модуль проецируется на этот же виртуальный адрес в процессе. В DLL содержатся *экспортируемые* (exported) функции, которые *импортирует* (import) клиентская программа (программа, загружающая DLL). Согласованием экспортированных и импортированных элементов занимается Windows.

Примечание DLL-модули в Win32 позволяют экспортировать глобальные переменные, не только функции.

В Win32 каждый процесс получает свою копию глобальных переменных DLL, доступных для чтения и записи. Чтобы несколько процессов совместно использовали какой-то участок памяти, надо либо прибегнуть к файлу, проецируемому в память, либо объявить *общий раздел данных* (shared data section), как описано в книге Джеффри Рихтера. Всякий раз, когда DLL запрашивает память из кучи, эта память выделяется из кучи, принадлежащей клиентскому процессу.

Согласование импортируемых элементов с экспортируемыми

DLL содержит таблицу экспортируемых функций. Эти функции идентифицируются по их символьному имени и (при необходимости) по целому числу — *порядковому номеру* (ordinal number). В таблице функций хранятся также адреса функций в пределах DLL. Впервые загружая DLL, клиентская программа не знает адресов нужных ей функций, зато знает их символьные имена или порядковые номера. Процесс динамической компоновки формирует таблицу, связывающую вызовы из клиентской программы с адресами функций в DLL. После модификации DLL собирать заново клиентскую программу не надо, если только вы не изменили имен функций или последовательности их параметров.

Примечание В простейшем случае вам хватило бы одного EXE-файла, импортирующего функции из одной или нескольких DLL. Но в реальности многие DLL в свою очередь вызывают функции из других DLL. Так что у каких-то DLL есть как экспортируемые, так и импортируемые элементы. Но это не проблема — механизм динамической компоновки справляется и с такими перекрестными связями.

В коде DLL экспортируемые функции надо объявлять явным образом, например:

```
__declspec(dllexport) int MyFunction(int n);
```

Альтернативный способ — перечислить экспортируемые функции в файле определения модуля (DEF) — гораздо хлопотнее. В клиентской программе надо указать, что функция импортируется:

```
__declspec(dllimport) int MyFunction(int n);
```

Однако компилятор C++ сгенерирует для *MyFunction* так называемое *расширенное имя* (decorated name), которое нельзя использовать в других языках. Это длинное имя составляется компилятором из имен класса и функции, а также типов параметров. Такие имена перечислены в MAP-файле проекта. Для упрощения имени функции, скажем, *MyFunction*, объявления следует писать так:

```
extern "C" __declspec(dllexport) int MyFunction(int n);
extern "C" __declspec(dllimport) int MyFunction(int n);
```

Примечание По умолчанию компилятор формирует передачу параметров по правилам *__cdecl*, а это значит, что параметры из стека выталкивает вызывающая программа. Программы на некоторых языках требуют придерживаться правил *__stdcall* (как в Pascal), когда стек очищает вызванная функция. Тогда в объявлении функции, экспортируемой из DLL, понадобится модификатор *__stdcall*.

Одних только объявлений импортируемых элементов недостаточно, чтобы клиент установил связь с DLL. В проекте клиентской программы надо определить *библиотеку импорта* (LIB) для компоновщика, а сама программа должна содержать *минимум один* явный вызов какой-то функции, импортируемой из DLL. Оператор вызова должен находиться на одном из путей выполнения программы[1].

Явное и неявное связывание

В предыдущем разделе в основном описывалось *неявное связывание* (implicit linking), которое программист на C++ скорее всего будет применять для своих DLL. Формируя DLL, компоновщик создает дополнительный LIB-файл, содержащий символьные имена всех элементов, экспортируемых из DLL и (при необходимости) их порядковые номера, но кода в нем нет. LIB-файл — это суррогат DLL, добавляемый к проекту клиентской программы. Когда собирается клиентская программа, импортируемые символьные имена согласуются с экспортируемыми из LIB, и эти имена (или порядковые номера) сохраняются в EXE-файле. LIB-файл содержит и имя DLL-файла (без указания пути), которое тоже хранится в EXE-файле. При загрузке клиента Windows находит и загружает нужные DLL-модули, а затем динамически связывает их по символьным именам или порядковым номерам.

Явное связывание (explicit linking) больше подходит для интерпретирующих языков вроде Microsoft JScript, но его можно использовать и в C++. При явном связывании файл импорта не нужен — вместо этого вы вызываете Win32-функцию *LoadLibrary*, указывая полное имя DLL как параметр. *LoadLibrary* возвращает параметр типа *HINSTANCE* — его можно использовать в вызове *GetProcAddress*, преобразующей символьное имя (или порядковый номер) в адрес. Пусть DLL экспортирует функцию так:

---

[1] Т. е. оператор вызова может и не выполняться, но должен находиться в таком месте программы, куда (пусть теоретически) может перейти управление. Только тогда компилятор гарантированно сгенерирует машинный код, реализующий вызов. — *Прим. перев.*

```
extern "C" __declspec(dllexport) double SquareRoot(double d);
```

Вот пример явного связывания клиента с экспортируемой функцией:

```
typedef double (SQRTPROC)(double);
HINSTANCE hInstance;
SQRTPROC* pFunction;
VERIFY(hInstance = ::LoadLibrary("c:\\winnt\\system32\\mydll.dll"));
VERIFY(pFunction = (SQRTPROC*)::GetProcAddress((HMODULE) hInstance, "SquareRoot"));
double d = (*pFunction)(81.0); // вызов функции из DLL
```

Если при явном связывании можно определить, когда загружать или выгружать DLL, то при неявном связывании все DLL загружаются в момент загрузки клиента. Явное связывание позволяет указывать и то, какие DLL следует загрузить. Например, пусть у вас есть одна DLL со строковыми ресурсами на английском языке, а другая — со строковыми ресурсами на русском. Когда пользователь выберет язык для работы, приложение загрузит соответствующую DLL.

Связывание по символьным именам и порядковым номерам

Связывание по порядковым номерам в Win16 было эффективнее и предлагалось по умолчанию. В Win32 связывание по символьным именам усовершенствовано, и теперь Microsoft рекомендует именно его. Однако в DLL-версии библиотеки MFC применяется связывание по порядковым номерам. Дело в том, что типичная программа на базе MFC подключается к сотням функций этой библиотеки и при связывании по порядковым номерам EXE-файл получается компактнее, так как отпадает необходимость хранить длинные символьные имена импортируемых элементов. Если вы создаете DLL со связыванием по порядковым номерам, определите в DEF-файле проекта номера, которые не слишком часто используются в Win32-среде. Если вы экспортируете функции C++, придется указывать в DEF-файле их расширенные имена (или объявить функции как *extern «C»*). Вот короткий фрагмент одного из DEF-файлов MFC-библиотеки:

```
??0CRecentFileList@@QAE@IPBDOHH@Z @ 479 NONAME
??0CRecordset@@QAE@PAVCDatabase@@@Z @ 480 NONAME
??0CRecordView@@IAE@I@Z @ 481 NONAME
??0CRecordView@@IAE@PBD@Z @ 482 NONAME
??0CRectTracker@@QAE@PBUtagRECT@@I@Z @ 483 NONAME
??0CReObject@@QAE@PAVCRichEditCntrItem@@@Z @ 484 NONAME
??0CReObject@@QAE@XZ @ 485 NONAME
??0CResetPropExchange@@QAE@XZ @ 486 NONAME
??0CRichEditCntrItem@@QAE@PAU_reobject@@PAVCRichEditDoc@@@Z @ 487 NONAME
??0CRichEditDoc@@IAE@XZ @ 488 NONAME
??0CRichEditView@@QAE@XZ @ 489 NONAME
??0CScrollView@@IAE@XZ @ 490 NONAME
??0CSemaphore@@QAE@JJPBDPAU_SECURITY_ATTRIBUTES@@@Z @ 491 NONAME
??0CSharedFile@@QAE@II@Z @ 492 NONAME
```

Числа после символов @ — это и есть порядковые номера. Ну что, вы по-прежнему хотите использовать символьные имена?

Точка входа в DLL: функция *DllMain*

По умолчанию компоновщик предполагает, что основная точка входа в DLL — функция *_DllMainCRTStartup*. Windows, загружая DLL, вызывает эту функцию, а та сначала вызывает конструкторы глобальных объектов, потом — глобальную функцию *DllMain*, которая, естественно, должна присутствовать. *DllMain* вызывается при подключении DLL к процессу, при отключении от него, а также в ряде других случаев. Взгляните на заготовку функции *DllMain*:

```
HINSTANCE g_hInstance;
extern "C" int APIENTRY
   DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      TRACE0("Ex20a.DLL Initializing!\n");
      // Do initialization here
   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      TRACE0("Ex20a.DLL Terminating!\n");
      // Do cleanup here
   }
   return 1; // ok
}
```

Если вы не пишете функцию *DllMain* для своей DLL, вместо нее подставляется версия-заглушка из стандартной библиотеки периода выполнения.

Функция *DllMain* вызывается также при запуске и завершении отдельных потоков, что определяет параметр *dwReason*. Впрочем, эта сложная тема исчерпывающе изложена в книге Джеффри Рихтера.

Описатели экземпляров и загрузка ресурсов

Каждая DLL в процессе идентифицируется уникальным 32-разрядным значением *HINSTANCE*. У процесса тоже есть описатель типа *HINSTANCE*. Все эти описатели экземпляров действительны только в рамках конкретного процесса и представляют собой начальный виртуальный адрес DLL- или EXE-модуля. В Win32 значения описателей *HINSTANCE* и *HMODULE* совпадают, и эти описатели равнозначны. Описатель экземпляра процесса почти всегда равен 0x400000, а у DLL с базовым адресом по умолчанию — 0x10000000. Если ваша программа работает с несколькими DLL, у них будут разные значения *HINSTANCE*, потому что в DLL указаны разные базовые адреса (определяется на этапе разработки) или потому что загрузчик скопировал и переместил код DLL.

Описатели экземпляров особенно важны при загрузке ресурсов. В частности, параметр *HINSTANCE* необходим Win32-функции *FindResource*. В EXE- и DLL-модулях могут быть свои ресурсы. Если нужен ресурс из DLL, указывают описатель экземпляра DLL, а если из EXE — описатель экземпляра EXE.

Как же получить описатель экземпляра? Вы уже видели, что описатель экземпляра DLL передается как параметр в функцию *DllMain*. Если вы хотите выяснить описатель EXE-модуля, вызовите Win32-функцию *GetModuleHandle* с параметром *NULL*. Если же нужен описатель DLL, вызовите *GetModuleHandle* с именем DLL в качестве параметра. Позже мы рассмотрим еще один метод загрузки ресурсов, реализованный в MFC-библиотеке, основанный на просмотре модулей в определенной последовательности.

Порядок поиска DLL клиентской программой

Если вы компонуете программу явным образом, используя *LoadLibrary*, то можете указывать полное имя DLL (с определением пути). Если же полный путь не указан, а также при неявной компоновке Windows будет искать вашу DLL в таком порядке.

1. В каталоге, содержащем данный EXE-файл.
2. В текущем каталоге процесса.
3. В системном каталоге Windows.
4. В каталоге Windows.
5. В каталогах, определенных в переменной окружения PATH.

Здесь есть одна ловушка, в которую можно запросто угодить. Вы создаете DLL как отдельный проект, затем копируете DLL-файл в системный каталог Windows и, наконец, загружаете DLL в клиентскую программу. До сих пор все было прекрасно. Но вот вы обновляете DLL, забываете скопировать ее в системный каталог, и при следующем запуске клиентской программы загружается старая версия DLL. Что тут сказать — будьте внимательны!

Отладка DLL

Visual C++ упрощает отладку DLL. Просто запустите отладчик в проекте DLL. При первом запуске отладчик запросит путь к клиентскому EXE-файлу. После этого при каждом «запуске» DLL из отладчика тот загружает этот EXE-файл, но в EXE-файл заложена своя последовательность поиска DLL-модулей. Это значит, что надо или настроить переменную окружения PATH, или скопировать DLL в каталог, который входит в последовательность поиска.

DLL-расширения и обычные DLL

До сих пор мы рассматривали DLL-модули Win32, в которых есть функция *DllMain* и какие-то экспортируемые функции. Теперь двинемся в мир, построенный на каркасе MFC-приложений, расширяющем базовую Win32-поддержку DLL. MFC Application Wizard позволяет создавать два вида DLL с поддержкой MFC-библиотеки: *DLL-расширения* (extension DLL) и *обычные DLL* (regular DLL). Но перед выбором надо вникнуть в суть предмета.

Примечание Конечно, Visual C++ .NET позволяет создавать Win32 DLL без библиотеки MFC — так же, как и программы. Но эта книга посвящена MFC, так что этот вариант нас не интересует.

DLL-расширение поддерживает интерфейс С++. Иначе говоря, такая DLL может экспортировать целые классы, а клиент может создавать объекты этих классов или разрабатывать производные классы. DLL-расширение динамически связывается с кодом DLL-версии библиотеки MFC, поэтому клиентская программа должна компоноваться с MFC тоже динамически (MFC Application Wizard предлагает этот вариант по умолчанию), а у клиентской программы и DLL-расширения должны совпадать версии динамических библиотек MFC (mfc42.dll, mfcd42.dll и т. д.). DLL-расширения весьма компактны: можно создать простое DLL-расширение размером всего 10 кб, которое загружается очень быстро.

Если же вам нужна DLL, которую можно загружать в любую Win32-среду программирования, тогда вам нужна обычная DLL. Однако здесь есть одно большое «но»: обычная DLL способна экспортировать только функции в стиле С и не поддерживает экспорт классов С++, функций-членов или переопределенных функций, ведь в каждом компиляторе С++ свой метод расширения имен функций. Правда, в самой DLL этого типа можно использовать классы С++ (в том числе классы MFC).

Создавая обычную DLL, вы можете решить, как связывать ее с MFC-библиотекой: статически или динамически. Если вы выберете первое, в DLL будет включена копия нужного кода MFC-библиотеки, и таким образом вы получите самодостаточный DLL-модуль. Средний размер статически связанной DLL без отладочной информации — около 144 кб. Если же вы предпочтете второе, размер уменьшится приблизительно до 17 кб, но при этом надо позаботиться о том, чтобы на компьютере пользователя присутствовали необходимые динамические модули MFC. Это не проблема, если клиентская программа уже динамически связана с той же версией MFC.

Когда вы сообщаете MFC Application Wizard, какой тип DLL или EXE вам нужен, *#define*-константы для компилятора устанавливаются так:

	Динамическое связывание с общей MFC-библиотекой	**Статическое связывание с MFC-библиотекой**
Обычная DLL	*_AFXDLL, _USRDLL*	*_USRDLL*
DLL-расширение	*_AFXEXT, _AFXDLL*	Не поддерживается
Клиентская EXE-программа	*_AFXDLL*	Константы не определены

Заглянув в исходный код и заголовочные MFC-файлы, вы увидите массу операторов *#ifdef* для этих констант. Это значит, что библиотечный код компилируется по-разному в зависимости от типа создаваемого проекта.

DLL-расширения: экспорт классов

Если DLL-расширение должно содержать только экспортируемые классы С++, создавать и использовать такую библиотеку легко. Описывая сборку Ex20a, мы расскажем, как сообщить MFC Application Wizard, что нужно создавать DLL-расширение. Генерируемая мастером заготовка содержит лишь функцию *DllMain*. Вы должны добавить в проект свои классы С++ и дополнить объявления классов макросом *AFX_EXT_CLASS*:

```
class AFX_EXT_CLASS CStudent : public CObject
```

Это изменение надо внести в заголовочный файл, часть DLL-проекта, и в аналогичный файл, используемый клиентской программой. Иначе говоря, заголовочные файлы клиента и DLL должны совпадать. Макрос генерирует код в зависимости от ситуации: класс экспортируется из DLL или импортируется в клиент.

Последовательность поиска ресурсов в DLL-расширении

Если вы создаете динамически связываемое *клиентское* приложение на базе MFC, многие стандартные ресурсы MFC-библиотеки (строки сообщений об ошибках, шаблоны диалоговых окон для предварительного просмотра перед печатью и др.) хранятся в DLL-модулях MFC, но и у вашего приложения есть свои ресурсы. Когда вы вызываете MFC-функцию вроде *CString::LoadString* или *CBitmap::LoadBitmap*, каркас приложений ищет сначала ресурсы EXE-файла, а потом ресурсы DLL MFC.

Если же программа включает в себя DLL-расширение, порядок поиска таков: сначала EXE-файл, затем DLL-расширение и, наконец, DLL MFC. При наличии, скажем, уникального идентификатора строкового ресурса MFC-библиотека найдет именно его, а если в EXE-файле и файле DLL-расширения какие-то идентификаторы строковых ресурсов дублируются, MFC-библиотека загрузит строку из EXE-файла.

Если же ресурс загружает DLL-расширение, последовательность поиска другая: сначала DLL-расширение, затем DLL MFC и, наконец, EXE-файл. При необходимости последовательность поиска можно изменить. Допустим, вы хотите в EXE-коде сначала искать ресурсы DLL-расширения. Это можно сделать так:

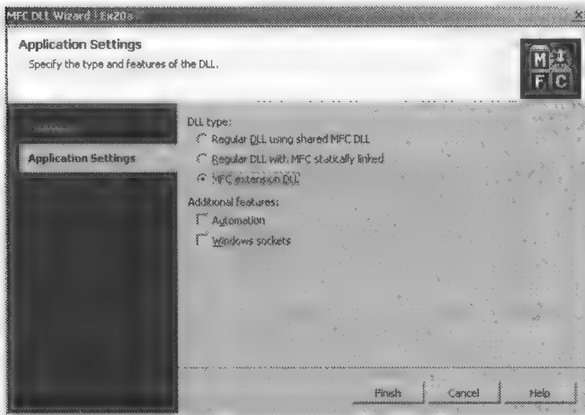
```
HINSTANCE hInstResourceClient = AfxGetResourceHandle();
// Используем описатель экземпляра DLL
AfxSetResourceHandle(::GetModuleHandle("mydllname.dll");
CString strRes;
StrRes.LoadString(IDS_MYSTRING);
// Восстанавливаем описатель экземпляра клиентской программы
AfxSetResourceHandle(hInstResourceClient);
```

Использовать здесь *AfxGetInstanceHandle* вместо *::GetModuleHandle* нельзя: в DLL-расширении *AfxGetInstanceHandle* возвращает описатель экземпляра EXE-, а не DLL-модуля.

Пример Ex20a: DLL-расширение

Мы превратим в DLL-расширение класс *CPersistentFrame* из главы 14. Сначала вы создадите файл Ex20a.dll, а потом используете его в клиентской программе Ex20b. Итак, создаем Ex20a.

1. **Средствами MFC Application Wizard создайте проект Ex20a.** Выберите в меню File последовательно команды New и Project. В качестве типа приложения выберите MFC DLL, а в качестве имени — Ex20a. На странице Application Settings мастера установите переключатель DLL type в положение MFC extension DLL:



2. **Просмотрите файл Ex20a.cpp.** MFC Application Wizard генерирует код, содержащий функцию *DllMain*:

```
// Ex20a.cpp : Defines the initialization routines for the DLL.
//
#include "stdafx.h"
#include <afxdllx.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

static AFX_EXTENSION_MODULE Ex20aDLL = { NULL, NULL };

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
   // Remove this if you use lpReserved
   UNREFERENCED_PARAMETER(lpReserved);

   if (dwReason == DLL_PROCESS_ATTACH)
   {
      TRACE0("Ex20a.DLL Initializing!\n");

      // Extension DLL one-time initialization
      if (!AfxInitExtensionModule(Ex20aDLL, hInstance))
         return 0;

(пропускаем сгенерированные строки комментариев)

      new CDynLinkLibrary(Ex20aDLL);

   }
   else if (dwReason == DLL_PROCESS_DETACH)
   {
      TRACE0("Ex20a.DLL Terminating!\n");
```

```
        // Terminate the library before destructors are called
        AfxTermExtensionModule(Ex20aDLL);
    }
    return 1; // ok
}
```

3. **Добавьте в проект класс *CPersistentFrame*.** Скопируйте файлы Persist.h и Persist.cpp из папки Ex14a на компакт-диске. Щелкните в меню Project команду Add Existing Item, в списке файлов выберите Persist.h и Persist.cpp и щелкните кнопку OK.
4. **Отредактируйте файл Persist.h.** Найдите строку:

```
class CPersistentFrame : public CFrameWnd
```

и поправьте ее:

```
class AFX_EXT_CLASS CPersistentFrame : public CFrameWnd
```

5. **Соберите проект и скопируйте файл DLL.** Скопируйте Ex20a.dll из каталога \vcppnet\Ex20a\Debug в системный каталог.

Пример Ex20b: тестовый клиент DLL-расширения

Эту программу мы начнем создавать как клиент для Ex20a.dll. Она импортирует из DLL класс *CPersistentFrame* и использует его как базовый класс окна-рамки SDI. Позже вы добавите в нее код, который позволит загружать и проверять другие примеры DLL из этой главы.

1. **С помощью MFC Application Wizard создайте проект Ex20b.** Это обычная EXE-программа на базе MFC. На странице Application Type мастера установите переключатель в положение Single document. Остальные параметры оставьте без изменения. Убедитесь, что переключатель Use of MFC на странице Application Type установлен в положение Use MFC in a shared DLL.
2. **Скопируйте файл Persist.h из каталога \vcppnet\Ex20a.** Заметьте: копировать надо заголовочный, а не CPP-файл.
3. **Замените базовый класс *CMainFrame* на *CPersistentFrame*, как это делалось в Ex14a.** Проведите глобальную замену *CFrameWnd* на *CPersistentFrame* как в MainFrm.h, так и в MainFrm.cpp, а также вставьте в MainFrm.h строку:

```
#include "persist.h"
```

4. **Добавьте библиотеку импорта Ex20a в список входных библиотек компоновщика.** Выберите в меню Project пункт Add Existing Item и в открывшемся окне найдите файл Ex20a.lib в каталоге \Ex20a\Debug на компакт-диске. Щелкните OK.
5. **Соберите и протестируйте программу Ex20b.** Если при запуске программы под отладчиком Windows не найдет Ex20a.dll, откроется окно с соответствующим сообщением. Если все пройдет благополучно, вы получите приложение с постоянной окном-рамкой, которое работает абсолютно так же, как и Ex14a. Все отличие в том, что код *CPersistentFrame* выделен в DLL-расширение.

Обычные DLL: структура *AFX_EXTENSION_MODULE*

Когда MFC Application Wizard генерирует обычную DLL, функция *DllMain* находится внутри каркаса приложений, а вы имеете дело с структурой типа *AFX_EXTENSION_MODULE* (и ее глобальным экземпляром). Во время инициализации DLL-расширений *AFX_EXTENSION_MODULE* служит для хранения состояния DLL- модуля.

Как правило, с этой структурой делать ничего не надо — вы можете создавать свои функции на C и экспортировать их модификатором *__declspec(dllexport)* (или создав записи в DEF-файле проекта).

Макрос *AFX_MANAGE_STATE*

Когда в процессе работы программы загружается mfc70.dll, она сохраняет данные в нескольких глобальных переменных. Если MFC-функции вызываются из MFC-программы или DLL-расширения, mfc70.dll «знает», как установить эти глобальные переменные для вызывающего процесса. Если же к mfc70.dll обратиться из обычной DLL, глобальные переменные не синхронизируются должным образом, и результат будет непредсказуем. Чтобы решить проблему, в начало каждой экспортируемой функции данной DLL надо вставить строку:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

Если же MFC-код связывается статически, макрос не оказывает никакого влияния.

Последовательность поиска ресурсов в обычной DLL

Когда EXE подключает обычную DLL, функции загрузки ресурсов в EXE-модуле загружают ресурсы самого EXE-модуля, а функции загрузки ресурсов в DLL — ресурсы этой DLL.

А вот чтобы код EXE-модуля загружал ресурсы из DLL, можно вызвать функцию *AfxSetResourceHandle* для временной смены описателя ресурса. Сам код практически не отличается от приведенного в разделе, посвященном поиску ресурсов в DLL-расширениях. Если вы пишете приложение, требующее локализации, то можете поместить зависящие от языка строки, диалоговые окна, меню и т. п. в обычные DLL (например, в English.dll, German.dll и French.dll). Клиентская программа явно загрузит нужную DLL и использует упомянутый выше код для загрузки ресурсов, которые, кстати, должны иметь одинаковые идентификаторы во всех DLL.

Пример Ex20c: обычная DLL

Мы создадим обычную DLL, которая экспортирует единственную функцию вычисления квадратного корня. Сначала мы соберем Ex20c.dll, а потом изменим клиентскую программу Ex20b, чтобы протестировать новую DLL.

1. **Средствами MFC Application Wizard создайте проект Ex20c.** Действуйте так же, как и в Ex20a, но в на странице Application Settings установите переключатель DLL type в положение Regular DLL Using Shared MFC DLL, а не MFC extension DLL.
2. **Просмотрите файл Ex20c.cpp.** MFC Application Wizard генерирует такой код:

```
// Ex20c.cpp : Defines the initialization routines for the DLL.
//
#include "stdafx.h"
#include "Ex20c.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

(пропускаем сгенерированные строки комментариев)

// CEx20cApp
BEGIN_MESSAGE_MAP(CEx20cApp, CWinApp)
END_MESSAGE_MAP()

// CEx20cApp construction
CEx20cApp::CEx20cApp()
{
   // TODO: add construction code here,
   // Place all significant initialization in InitInstance
}

// The one and only CEx20cApp object
CEx20cApp theApp;


// CEx20cApp initialization
BOOL CEx20cApp::InitInstance()
{
   CWinApp::InitInstance();

   return TRUE;
}
```

3. **Добавьте код экспортируемой функции *Ex20cSquareRoot*.** Этот код можно ввести как в Ex20c.cpp, так и в новый файл:

```
extern "C" __declspec(dllexport) double Ex20cSquareRoot(double d)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   TRACE("Entering Ex20cSquareRoot\n");
   if (d >= 0.0) {
      return sqrt(d);
   }
   AfxMessageBox("Can't take square root of a negative number.");
   // Нельзя извлечь корень из отрицательного числа
   return 0.0;
}
```

Как видите, проблем с выводом окон сообщений и других модальных диалоговых окон в DLL нет. Не забудьте включить math.h в файл с кодом функции

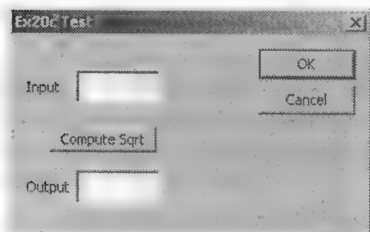
Ex20cSquareRoot. Кроме того, укажите прототип функции *Ex20cSquareRoot* в файле Ex20c.h, чтобы она была видна внешним клиентам.

4. **Соберите проект и скопируйте файл DLL.** Скопируйте Ex20c.dll из каталога \Ex20c\Debug в системный каталог.

Коррекция Ex20b для проверки Ex20c.dll

Когда вы собирали программу Ex20b, она динамически связывалась с DLL-расширением Ex20a. Теперь мы изменим проект, чтобы неявно скомпоновать Ex20b с обычной DLL (Ex20c) и вызывать из нее функцию квадратного корня.

1. **Добавьте новый диалоговый ресурс и класс в Ex20b.** С помощью редактора диалоговых окон создайте шаблон *IDD_EX20C*:



Затем средствами мастера Add Class Wizard сгенерируйте класс *CTest20cDialog*, производный от *CDialog*. Элементы управления, переменные-члены и функция карты сообщений описаны в таблице:

Идентификатор элемента управления	Тип	Переменная-член	Функция карты сообщений
IDC_INPUT	Поле ввода	*m_dInput (double)*	
IDC_OUTPUT	Поле ввода	*m_dOutput (double)*	
IDC_COMPUTE	Кнопка		*OnBnClickedCompute*

2. **Напишите код функции *OnBnClickedCompute*, чтобы вызывать экспортируемую функцию DLL.** Отредактируйте заготовку функции в Test20cDialog.cpp:

```
void CTest20cDialog::OnBnClickedCompute()
{
    UpdateData(TRUE);
    m_dOutput = Ex20cSquareRoot(m_dInput);
    UpdateData(FALSE);
}
```

Объявите *Ex20cSquareRoot* как импортируемую. Для этого добавьте в Test20cDialog.h:

```
extern "C" __declspec(dllimport) double Ex20cSquareRoot(double d);
```

3. **Введите класс *CTest20cDialog* в приложение Ex20b.** Добавьте меню верхнего уровня Test и команду Ex20c DLL с идентификатором *ID_TEST_EX20CDLL*. В окне Properties утилиты Class View сопоставьте эту команду функции-члену класса *CEx20bView* и напишите код обработчика в Ex20bView.cpp:

```
void CEx20bView::OnTestEx20cdll()
{
    CTest20cDialog dlg;
    dlg.DoModal();
}
```

И, конечно же, дополните файл Ex20bView.cpp строкой:

```
#include "Test20cDialog.h"
```

4. **Добавьте библиотеку импорта Ex20c в список входных библиотек компоновщика.** Выберите меню Project в Visual C++ .NET команду Add Existing Item и добавьте в проект файл \Ex20c\Debug\Ex20c.lib. Теперь программа будет неявно компоноваться с Ex20a.DLL и Ex20c.DLL. Как видите, клиенту, в общем-то, все равно, какая это DLL: обычная или DLL-расширение. Вы лишь указываете компоновщику имя библиотеки LIB.
5. **Соберите и протестируйте измененное приложение Ex20b.** Выберите в меню Test команду Ex20c DLL. Введите в поле ввода Input какое-нибудь число и щелкните кнопку Compute Sqrt. Результат должен появиться в поле Output.

DLL с пользовательскими элементами управления

Программисты применяют DLL для хранения *пользовательских элементов управления* (custom controls) чуть ли не с первых дней Windows, потому что такие элементы управления полностью автономны. Раньше их писали на С и конфигурировали как автономные DLL. Сегодня к нашим услугам MFC-библиотеки и мастера, позволяющие упростить программирование. Для пользовательских элементов управления лучше всего подходит обычная DLL, поскольку им не нужен интерфейс C++ и их предполагается использовать в любой среде программирования, воспринимающей такого рода элементы (например, в Borland C++). Кроме того, вы, наверное, предпочтете вариант динамической компоновки с MFC — в этом случае DLL компактнее и загружается быстрее.

Понятие пользовательского элемента управления

Вы уже познакомились с обычными (в главе 7) и стандартными (в главе 8) элементами управления Windows, а в главе 9 — и с ActiveX-элементами. Пользовательский элемент управления аналогичен обычному (вроде поля ввода) в том плане, что тоже посылает родительскому окну уведомляющие сообщения *WM_COMMAND* и получает сообщения, определяемые пользователем. Редактор диалоговых окон позволяет включать такие элементы в шаблоны диалоговых окон. Для этого и предназначена соответствующая кнопка в окне палитры элементов управления.

Вам предоставляется большая свобода при разработке собственного элемента управления. Вы можете нарисовать в его окне, управляемом клиентским приложением, все, что вздумается, и определить любое уведомление и связанные с ним сообщения. Соотнести обычные Windows-сообщения со своим элементом управления (например, *WM_LBUTTONDOWN*) позволяет окно Properties утилиты Class View,

но обработчики уведомляющих сообщений в классе родительского окна и обработчики пользовательских сообщений придется создавать вручную.

Оконный класс пользовательского элемента управления

Пользовательские элементы управления определяются в шаблоне диалогового ресурса по символьным именам их *оконных классов*. Не путайте оконный класс Win32 с классом C++ — общее у них только название. Оконный класс определяет структура, содержащая:

- имя класса;
- указатель на функцию *WndProc*, которая принимает сообщения, отправленные в окна данного класса;
- вспомогательные атрибуты (например, кисть фона).

Win32-функция *RegisterClass* копирует структуру в память процесса так, что любая функция в данном процессе может создать окно на основе этого класса. После инициализации диалогового окна Windows создает дочерние окна пользовательских элементов управления в соответствии с именами оконных классов, хранящихся в шаблоне.

Пусть функция *WndProc* элемента управления находится в DLL. При инициализации (вызовом *DllMain*) DLL может вызвать для элемента управления функцию *RegisterClass*. Поскольку DLL является частью процесса, клиентская программа получает возможность создавать дочерние окна класса пользовательского элемента управления. Клиент знает имя оконного класса элемента управления и использует его при создании дочернего окна. Весь код элемента управления, включая *WndProc*, хранится в DLL. Вам нужно только, чтобы клиент загружал DLL до создания дочернего окна.

Библиотека MFC и функция *WndProc*

Итак, Windows вызывает *WndProc* элемента управления для каждого сообщения, отправленного в это окно. Но не пишите старомодные операторы *switch/case* — сопоставьте эти сообщения с функциями-членами классов C++ так, как вы уже делали это ранее. Для этого добавьте в DLL класс C++, соответствующий оконному классу элемента управления; затем в окне Properties утилиты Class View создайте обработчики сообщений.

Написать класс C++ для элемента управления несложно — просто создайте с помощью Add Class Wizard новый класс, производный от *CWnd*. Куда сложнее связать класс C++ с функцией *WndProc* и системой маршрутизации сообщений в каркасе приложений. В примере Ex20d вы увидите реальную функцию *WndProc*, а здесь мы приведем псевдокод типичной функции элемента управления:

```
LRESULT MyControlWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   if (это первое сообщение для данного окна) {
      CWnd* pWnd = new CMyControlWindowClass();
      свяжите pWnd с hWnd
   }
   return AfxCallWndProc(pWnd, hWnd, message, wParam, lParam);
}
```

MFC-функция *AfxCallWndProc* передает сообщения каркасу приложений, а тот пересылает их функциям-членам *CMyControlWindowClass*.

Уведомляющие сообщения пользовательских элементов управления

Элемент управления общается с родительским окном, посылая ему уведомляющие сообщения *WM_COMMAND* с такими параметрами:

Параметр	Назначение
wParam (старшее «слово»)	Код уведомления
wParam (младшее «слово»)	Идентификатор дочернего окна
lParam	Описатель дочернего окна

Значение кода уведомления зависит от конкретного элемента управления. Родительское окно должно «уметь» интерпретировать этот код с учетом того, что ему известно о данном элементе управления. Например, код 77 мог бы означать, что пользователь ввел символ при установленном на элементе управления фокусе ввода. Элемент управления может отправить уведомляющее сообщение так:

```
GetParent()->SendMessage(WM_COMMAND,
                GetDlgCtrlID() | ID_NOTIFYCODE << 16, (LONG) GetSafeHwnd());
```

На клиентской стороне нужно сопоставить это сообщение функции-обработчику посредством MFC-макроса *ON_CONTROL*:

```
ON_CONTROL(ID_NOTIFYCODE, IDC_MYCONTROL, OnClickedMyControl)
```

и объявить функцию-обработчик:

```
afx_msg void OnClickedMyControl();
```

Пользовательские сообщения, направляемые в элемент управления

Вы уже сталкивались с пользовательскими сообщениями в главе 7. Они нужны клиентской программе для взаимодействия с элементом управления. Поскольку стандартное сообщение возвращает 32-разрядное значение (если сообщение отправлено синхронно), в ответ клиент может получать информацию от элемента управления.

Пример Ex20d: пользовательский элемент управления

Ex20d — это обычная DLL на базе MFC, которая реализует элемент управления «светофор» и переключает его состояние: «выключен», красный, желтый, зеленый. При щелчке светофора левой кнопкой мыши уведомляется его родительское окно, кроме того, он реагирует на два пользовательских сообщения: *RYG_SETSTATE* и *RYG_GETSTATE*. Состояние определяется целым числом, кодирующим цвет. Автор этого элемента управления — Ричард Уилтон (Richard Wilton) — включил его исходную версию на языке С в книгу «Windows 3 Developer's Workshop» (Microsoft Press, 1991).

Изначально проект Ex20d сгенерирован MFC Application Wizard (компоновка с общими DLL библиотеки MFC) — как и Ex20c. Исходный текст показан ниже, причем добавленный в функцию *InitInstance* код выделен. Экспортируемая функция-заглушка *Ex20dEntry* нужна лишь для того, чтобы DLL можно было подключить неявно. Клиентская программа должна вызывать эту функцию. Вызов должен быть в пути исполнения программы, иначе компилятор проигнорирует вызов. В качестве альтернативы клиентская программа могла бы подключать DLL и явно, вызывая в *InitInstance* Win32-функцию *LoadLibrary*.

Ex20d.cpp

```
// Ex20d.cpp : Defines the initialization routines for the DLL.
//
#include "stdafx.h"
#include "Ex20d.h"

#include "rygwnd.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

extern "C" __declspec(dllexport) void Ex20dEntry() {} // функция-заглушка
// [illegible]
[illegible]
// [illegible]

BEGIN_MESSAGE_MAP(CEx20dApp, CWinApp)
END_MESSAGE_MAP()

// CEx20dApp construction
CEx20dApp::CEx20dApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

// The one and only CEx20dApp object
CEx20dApp theApp;

// CEx20dApp initialization
BOOL CEx20dApp::InitInstance()
{
    CRygWnd::RegisterWndClass(AfxGetInstanceHandle());
    CWinApp::InitInstance();

    return TRUE;
}
```

Ниже показан листинг класса *CRygWnd*, в том числе глобальной функции *RygWndProc*. Для создания этого класса вызовите мастер Add Class Wizard в меню Project

выберите Add Class. Код изменения цвета светофора не очень интересен, поэтому мы сосредоточимся на функциях, общих для большинства пользовательских элементов управления. Статическая функция-член *RegisterWndClass* регистрирует оконный класс *RYG* и должна вызываться сразу после загрузки DLL. Обработчик *OnLButtonDown* вызывается при щелчке левой кнопкой мыши в окне элемента управления. Он отправляет уведомление о щелчке родительскому окну. Переопределенная функция *PostNcDestroy* очень важна — она удаляет объект *CRygWnd*, когда клиентская программа уничтожает окно элемента управления. Функции *OnGetState* и *OnSetState* вызываются в ответ на пользовательские сообщения, синхронно отправленные клиентом. И последнее: не забудьте скопировать DLL в системный каталог.

RygWnd.h

```
[illegible]

#define RYG_SETSTATE WM_USER + 0
#define RYG_GETSTATE WM_USER + 1

LRESULT CALLBACK AFX_EXPORT
    RygWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
[illegible]
[illegible]
[illegible]

private:
    int m_nState; // 0="выключен", 1=красный, 2=желтый, 3=зеленый

    static CRect  s_rect;
    static CPoint s_point;
    static CRect  s_rColor[3];
    static CBrush s_bColor[4];

public:
    static BOOL RegisterWndClass(HINSTANCE hInstance);
    [illegible]

[illegible]
    [illegible]
        [illegible]

private:
    void SetMapping(CDC* pDC);
    void UpdateColor(CDC* pDC, int n);
[illegible]
    afx_msg LRESULT OnSetState(WPARAM wParam, LPARAM lParam);
    afx_msg LRESULT OnGetState(WPARAM wParam, LPARAM lParam);
    [illegible]

[illegible]
```

RygWnd.cpp

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]

LRESULT CALLBACK AFX_EXPORT
   RygWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   CWnd* pWnd;

   pWnd = CWnd::FromHandlePermanent(hWnd);
   if (pWnd == NULL) {
      // Предполагаем, что клиент создал окно CRygWnd
      pWnd = new CRygWnd();
      pWnd->Attach(hWnd);
   }
   ASSERT(pWnd->m_hWnd == hWnd);
   ASSERT(pWnd == CWnd::FromHandlePermanent(hWnd));
   LRESULT lResult = AfxCallWndProc(pWnd, hWnd, message,
                                    wParam, lParam);
   return lResult;
}


// статические переменные-члены
CRect  CRygWnd::s_rect(-500, 1000, 500, -1000); // ограничивающий  прямоугольник
CPoint CRygWnd::s_point(300, 300); // скругленные  углы
CRect  CRygWnd::s_rColor[] = {CRect(-250, 800, 250, 300),
                              CRect(-250, 250, 250, -250),
                              CRect(-250, -300, 250, -800)};
CBrush CRygWnd::s_bColor[] = {RGB(192, 192, 192),
                              RGB(0xFF, 0x00, 0x00),
                              RGB(0xFF, 0xFF, 0x00),
                              RGB(0x00, 0xFF, 0x00)};

BOOL CRygWnd::RegisterWndClass(HINSTANCE hInstance) // статическая  функция-член
{
   WNDCLASS wc;
   wc.lpszClassName = "RYG"; // соответствует имени класса в клиенте
   wc.hInstance = hInstance;
   wc.lpfnWndProc = RygWndProc;
   wc.hCursor = ::LoadCursor(NULL, IDC_ARROW);
   wc.hIcon = 0;
   wc.lpszMenuName = NULL;
```

см. след. стр.

```
    wc.hbrBackground = (HBRUSH) ::GetStockObject(LTGRAY_BRUSH);
    wc.style = CS_GLOBALCLASS;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    return (::RegisterClass(&wc) != 0);
}
[illegible]
[illegible]
[illegible]
{
    m_nState = 0;
    TRACE("CRygWnd constructor\n");
}

[illegible]
{
    TRACE("CRygWnd destructor\n");
}

[illegible]
    ON_MESSAGE(RYG_SETSTATE, OnSetState)
    ON_MESSAGE(RYG_GETSTATE, OnGetState)
    [illegible]
    [illegible]
[illegible]

void CRygWnd::SetMapping(CDC* pDC)
{
    CRect clientRect;
    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ISOTROPIC);
    pDC->SetWindowExt(1000, 2000);
    pDC->SetViewportExt(clientRect.right, -clientRect.bottom);
    pDC->SetViewportOrg(clientRect.right / 2, clientRect.bottom / 2);
}

void CRygWnd::UpdateColor(CDC* pDC, int n)
{
    if (m_nState == n + 1) {
        pDC->SelectObject(&s_bColor[n+1]);
    }
    else {
        pDC->SelectObject(&s_bColor[0]);
    }
    pDC->Ellipse(s_rColor[n]);
}
[illegible]
[illegible]
{
    int i;
```

```
    CPaintDC dc(this); // контекст устройства для рисования
    SetMapping(&dc);
    dc.SelectStockObject(DKGRAY_BRUSH);
    dc.RoundRect(s_rect, s_point);
    for (i = 0; i < 3; i++) {
        UpdateColor(&dc, i);
    }
}

void CRygWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
    // код уведомления в HIWORD wParam, в данном случае равен 0
    GetParent()->SendMessage(WM_COMMAND, GetDlgCtrlID(),
        (LONG) GetSafeHwnd()); // 0
}

void CRygWnd::PostNcDestroy()
{
    TRACE("CRygWnd::PostNcDestroy\n");
    delete this; // CWnd::PostNcDestroy ничего не делает
}

LRESULT CRygWnd::OnSetState(WPARAM wParam, LPARAM lParam)
{
    TRACE("CRygWnd::SetState, wParam = %d\n", wParam);
    m_nState = (int) wParam;
    Invalidate(FALSE);
    return 0L;
}

LRESULT CRygWnd::OnGetState(WPARAM wParam, LPARAM lParam)
{
    TRACE("CRygWnd::GetState\n");
    return m_nState;
}
```

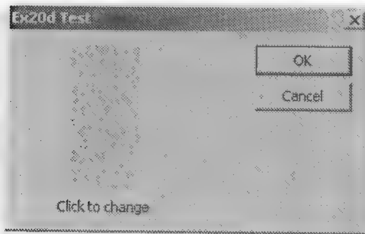
Коррекция Ex20b для проверки Ex20d.dll

Программа Ex20b уже компонуется с DLL-модулями Ex20a и Ex20c. Теперь мы переработаем проект так, чтобы программу неявно связать с пользовательским элементом управления Ex20d.

1. **Добавьте новый диалоговый ресурс и класс в проект Ex20b.** С помощью редактора диалоговых окон создайте шаблон *IDD_EX20D* с пользовательским элементом управления и идентификатором дочернего окна *IDC_RYG*.

   Укажите для оконного класса пользовательского элемента управления имя *RYG*.

   Затем средствами мастера Add Class Wizard в окне Properties утилиты Class View сгенерируйте класс *CTest20dDialog*, производный от *CDialog*.



2. **Отредактируйте файл Test20dDialog.h.** Добавьте закрытую переменную-член:

```
enum {OFF, RED, YELLOW, GREEN} m_nState;
```

Объявите также импортируемую функцию и идентификаторы пользовательских сообщений:

```
extern "C" __declspec(dllimport) void Ex21dEntry(); // функция-заглушка
#define RYG_SETSTATE WM_USER + 0
#define RYG_GETSTATE WM_USER + 1
```

3. **Отредактируйте конструктор в Test20dDialog.cpp для инициализации переменной-члена – флажка состояния.** Добавьте выделенный код:

```
CTest20dDialog::CTest20dDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CTest21dDialog::IDD, pParent)
{
    m_nState = OFF;
    Ex20dEntry(); // Проверяем загрузку DLL
}
```

4. **Создайте обработчик уведомляющего сообщения о щелчке элемента управления.** Здесь Class View не поможет — придется вручную добавить элемент таблицы сообщений и функцию-обработчик в файл Test20dDialog.cpp:

```
void CTest20dDialog::OnClickedRyg()
{
    switch(m_nState) {
    case OFF:
        m_nState = RED;
        break;
    case RED:
        m_nState = YELLOW;
        break;
    case YELLOW:
        m_nState = GREEN;
        break;
    case GREEN:
        m_nState = OFF;
        break;
    }
    GetDlgItem(IDC_RYG)->SendMessage(RYG_SETSTATE, m_nState);
    return;
}
```

```
BEGIN_MESSAGE_MAP(CTest20dDialog, CDialog)
   ON_CONTROL(0, IDC_RYG, OnClickedRyg) // код уведомления - 0
END_MESSAGE_MAP()
```

Получив уведомление о щелчке, диалоговое окно отправляет сообщение *RYG_SETSTATE* обратно элементу управления, чтобы тот изменил свой цвет. Не забудьте добавить в файл Test20dDialog.h прототип:

```
afx_msg void OnClickedRyg();
```

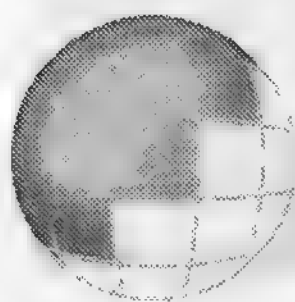
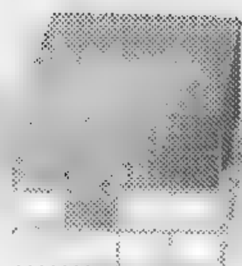
5. **Включите класс *CTest20dDialog* в приложение Ex20b.**
6. **Добавьте в меню Test вторую команду — Ex20d DLL с идентификатором *ID_TEST_EX20DDLL*.** В окне Properties утилиты Class View сопоставьте эту команду функции-члену в классе *CEx20bView* и напишите код обработчика в Ex20bView.cpp:

```
void CEx20bView::OnTestEx20ddll()
{
   CTest20dDialog dlg;
   dlg.DoModal();
}
```

И, конечно, включите в файл Ex20bView.cpp строку:

```
#include "Test20dDialog.h"
```

7. **Добавьте библиотеку импорта Ex20d в список входных библиотек компоновщика.** В меню Project выберите команду Add Existing Item и добавьте в проект файл \vcppnet\ Ex20d\Debug\Ex20.lib. Теперь программа должна неявно подключать все три DLL.
8. **Соберите и протестируйте измененную программу Ex20b.** Выберите из меню Test команду Ex20d DLL. Попробуйте пощелкать светофор левой кнопкой мыши — он должен изменять свой цвет.

ГЛАВА

21

MFC-программы без классов «документ» и «вид»

Архитектура «документ-вид» полезна для создания многих приложений, но иногда можно обойтись более простой структурой программы. В этой главе приведены три приложения, основанные: на диалоговом окне, SDI- и MDI-интерфейсах. Ни в одном из них нет классов «документ», «вид» или «шаблон документа», зато есть система маршрутизации команд и ряд других средств библиотеки MFC. В Visual C++ .NET все три типа приложений создаются средствами MFC Application Wizard.

В каждом примере мы посмотрим, как MFC Application Wizard генерирует код, не зависящий от архитектуры «документ-вид», и покажем, как добавлять в приложение свой код.

Пример Ex21a: приложение — диалоговое окно

Во многих приложениях пользовательский интерфейс вполне может состоять из диалогового окна — оно открывается после запуска приложения. Пользователь может свернуть его и, поскольку оно не системное модальное, свободно переключаться на другие программы.

В этом примере диалоговое окно — это простой калькулятор (рис. 21-1). Class View определяет переменные-члены класса и создает вызовы DDX-функций, отвечающих за обмен данными в диалоговых окнах, — словом, возьмет на себя все, кроме кодирования функции, нужной для вычислений. Диалоговое окно и значок программы определены в файле описания ресурсов Ex21a.rc.

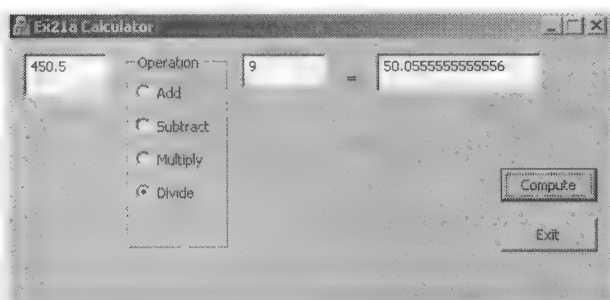
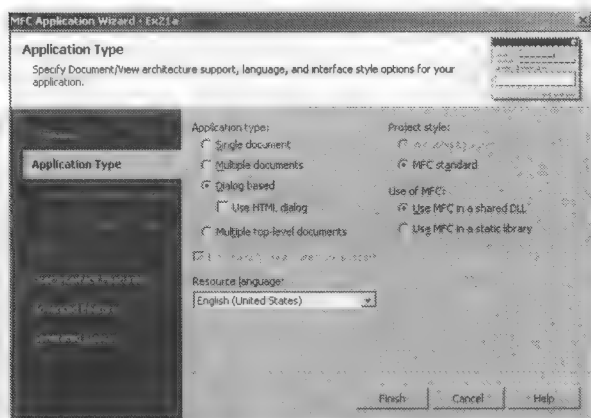


Рис. 21-1. *Диалоговое окно Ex21a Calculator*

MFC Application Wizard поддерживает создание приложений на основе диалогового окна, чем мы и воспользуемся.

1. **С помощью MFC Application Wizard создайте проект Ex21a.** На странице Application Type установите переключатель в положение Dialog Based.



   На странице User Interface Features в поле Dialog title введите заголовок окна — **Ex21a Calculator**.

2. **Отредактируйте ресурс *IDD_EX21A_DIALOG*.** При этом руководствуйтесь рис. 21-1. Назначить элементам управления идентификаторы (см. таблицу) поможет редактор диалоговых окон. Затем откройте окно Properties диалогового окна и присвойте свойствам System Menu и Minimize Box значение TRUE.

Элемент управления	Идентификатор
Поле ввода левого операнда	*IDC_LEFT*
Поле ввода правого операнда	*IDC_RIGHT*
Поле ввода для результата	*IDC_RESULT*
Первый переключатель (с группированием)	*IDC_OPERATION*
Кнопка Compute	*IDC_COMPUTE*

3. **Средствами Add Member Variable Wizard добавьте переменные-члены и в окне Properties утилиты Class View создайте обработчик команды.**

MFC Application Wizard уже сгенерировал класс *CEx21aDlg*. Дополните его переменными-членами:

Идентификатор	Переменные-члены	Тип
IDC_LEFT	*m_dLeft*	Double
IDC_RIGHT	*m_dRight*	Double
IDC_RESULT	*m_dResult*	Double
IDC_OPERATION	*m_nOperation*	int

Добавьте обработчик сообщений *OnBnClickedCompute* для кнопки *IDC_COMPUTE*.

4. **Напишите функцию-член *OnBnClickedCompute* в файле Ex21aDlg.cpp.** Введите выделенный код:

```
void CEx21aDlg::OnBnClickedCompute()
{
    UpdateData(TRUE);
    if(m_nOperation == 0) {
        m_dResult = m_dLeft + m_dRight;
    } else if(m_nOperation == 1) {
        m_dResult = m_dLeft - m_dRight;
    } else if(m_nOperation == 2) {
        m_dResult = m_dLeft * m_dRight;
    } else if(m_nOperation == 3) {
        if(m_dRight == 0) {
            AfxMessageBox("Divide by zero");
        } else {
            m_dResult = m_dLeft / m_dRight;
        }
    }
    UpdateData(FALSE);
}
```

5. **Соберите и протестируйте приложение Ex21a.** Заметьте: значок программы появляется на панели задач Windows. Убедитесь, что диалоговое окно можно свернуть.

Функция *InitInstance* класса приложения

Важный элемент приложения Ex21a — функция *CEx21aApp::InitInstance*, созданная MFC Application Wizard. Обычная функция *InitInstance* создает основное окно-рамку и возвращает *TRUE*, после чего запускается цикл обработки сообщений. Но ее версия в Ex21a конструирует объект модального диалогового окна, вызывает *DoModal* и возвращает *FALSE*. Это означает, что приложение завершается, как только пользователь закрывает диалоговое окно. Функция *DoModal* позволяет Windows-процедуре диалогового окна получать и распределять сообщения как обычно. Заметьте: MFC Application Wizard не создает вызова *CWinApp::SetRegistryKey*.

Взгляните на сгенерированный код *InitInstance* из Ex21a.cpp:

```
BOOL CEx21aApp::InitInstance()
{
    // InitCommonControls() is required on Windows XP if an application
```

```
    // manifest specifies use of ComCtl32.dll version 6 or later to enable
    // visual styles.  Otherwise, any window creation will fail.
    InitCommonControls();
    CWinApp::InitInstance();
    AfxEnableControlContainer();

    CEx21aDlg dlg;
    m_pMainWnd = &dlg;
    INT_PTR nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
       // TODO: Place code here to handle when the dialog is
       //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
       // TODO: Place code here to handle when the dialog is
       //  dismissed with Cancel
    }
    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}
```

Класс диалогового окна и значок программы

В созданном мастером классе *CEx21aDlg* есть два элемента таблицы сообщений:

```
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
```

Соответствующие функции-обработчики отвечают за отображение значка программы при сворачивании ее окна. Они нужны только в Windows NT 3.51, в которой значки свернутых программ находятся на рабочем столе, и не требуются ни в Windows 95/98, ни в Windows NT 4.0/2000/XP, поскольку в них значки свернутых программ размещаются на панели задач.

Однако кое-какой код для значков нужен. Он находится в сгенерированном MFC Application Wizard обработчике *OnInitDialog* сообщения *WM_INITDIALOG*. Обратите внимание на два вызова *SetIcon*. MFC Application Wizard формирует код для добавления в системное меню команды About (если вы установили флажок About box). *m_hIcon* — переменная-член класса диалогового окна, инициализируемая в конструкторе.

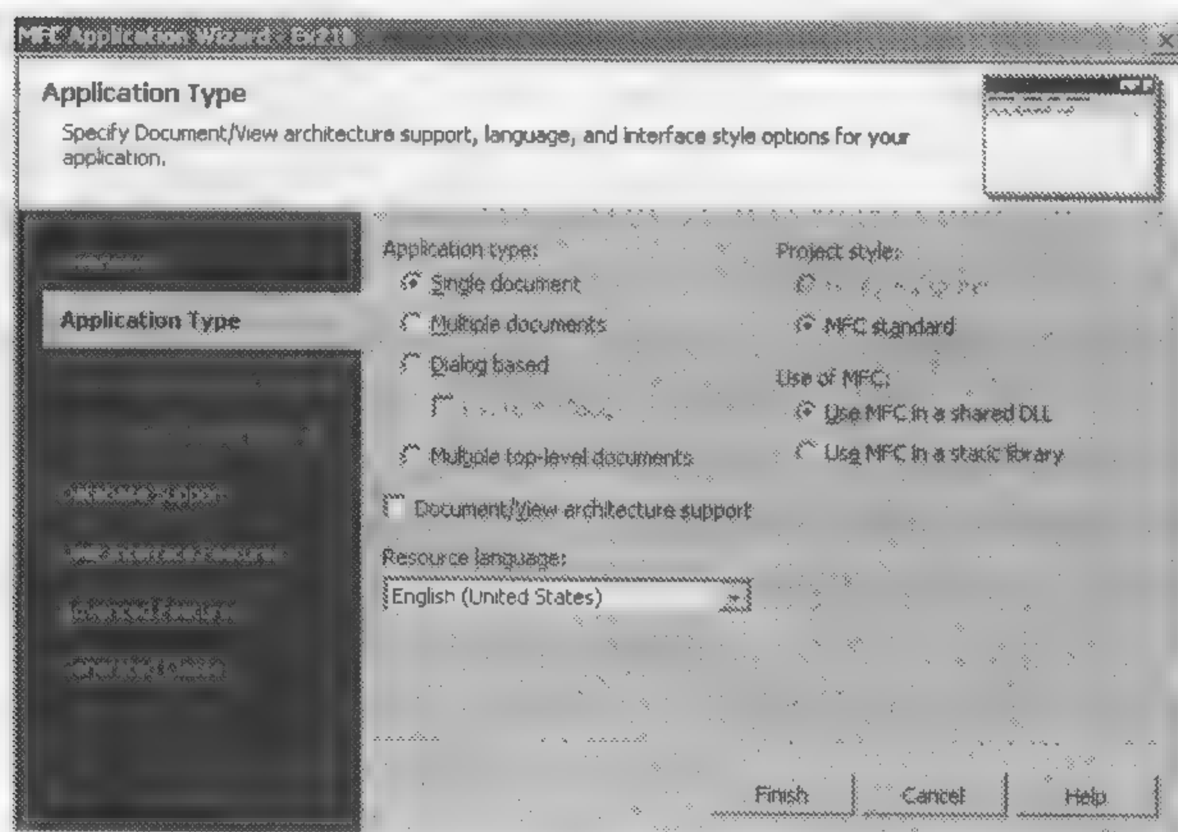
```
BOOL CEx21aDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Add "About..." menu item to system menu.
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
```

```
    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
       CString strAboutMenu;
       strAboutMenu.LoadString(IDS_ABOUTBOX);
       if (!strAboutMenu.IsEmpty())
       {
          pSysMenu->AppendMenu(MF_SEPARATOR);
          pSysMenu->AppendMenu(MF_STRING,
                               IDM_ABOUTBOX, strAboutMenu);
       }
    }
    // Set the icon for this dialog.  The framework does this
    //  automatically when the application's main window
    //  is not a dialog.
    SetIcon(m_hIcon, TRUE);     // Set big icon
    SetIcon(m_hIcon, FALSE);      // Set small icon
    // TODO: Add extra initialization here
    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Пример Ex21b: SDI-программа

Эта SDI-программа из серии «Hello, world!» построена на основе кода из главы 2. У нее одно окно — объект класса, наследующего классу *CFrameWnd*. Операции прорисовки выполняются в окне-рамке, там же обрабатываются и все сообщения.

1. **Средствами MFC Application Wizard Ex21b.** На странице Application Type установите переключатель в положение Single document и сбросьте флажок Document/View Architecture Support:



2. **Добавьте код прорисовки диалогового окна.** В функцию *CChildView::OnPaint* в файле ChildView.cpp добавьте выделенный код:

```
void CChildView::OnPaint()
{
```

```
    CPaintDC dc(this); // device context for painting

    dc.TextOut(0, 0, "Hello, world!");

    // Do not call CWnd::OnPaint() for painting messages
}
```

3. **Скомпилируйте и запустите приложение.** Мы получили полноценное SDI-приложение, которое никак не зависит от архитектуры «документ-вид».

MFC Application Wizard автоматически удаляет из приложения все следы этой зависимости и создает следующие элементы.

- **Основное меню.** Windows-приложение может обойтись без меню и даже без описания ресурсов. Но в примере Ex21b есть и то и другое. Каркас приложений маршрутизирует команды меню обработчикам сообщений в классе окна-рамки.
- **Значок.** Полезен, если программу предполагается запускать из Windows Explorer или сворачивать ее основное окно-рамку. Значок, как и меню, хранится в ресурсе.
- **Обработчик командного сообщения о закрытии окна.** Многие программы должны проделывать ряд особых операций в момент закрытия основного окна. Если бы вы использовали документы, то могли бы переопределить функцию *CDocument::SaveModified.* Но здесь, чтобы контролировать процесс закрытия, надо написать обработчики сообщений о закрытии окна, посылаемых программе в ответ на действия пользователя или самой Windows при завершении ее работы.
- **Панель инструментов и строка состояния.** MFC Application Wizard автоматически генерирует панель инструментов и строку состояния и настраивает маршрутизацию сообщений, хотя классов «документ» и «вид» здесь нет.

  У SDI-приложений без поддержки архитектуры «документ-вид» есть несколько интересных характерных особенностей.

- **Класс *CChildView*** вопреки своему названию на самом деле наследует *CWnd* и объявляется в ChildView.h и реализуется в ChildView.cpp. Этот класс реализует только виртуальную функцию-член *OnPaint*, которая содержит весь код для рисования в окне-рамке всего, что нужно (см. шаг 2 в примере Ex21b.)
- **Класс *CMainFrame*** содержит переменную-член *m_wndView*, которая создается и инициализируется в функции *CMainFrame::OnCreate.*
- **Функция *CMainFrame::OnSetFocus*** передает фокус окну *CChildView*:

```
void CMainFrame::OnSetFocus(CWnd* pOldWnd)
{
    // передаем фокус дочернему окну
    m_wndView.SetFocus();
}
```

- **Функция *CMainFrame::OnCmdMsg*** дает шанс дочернему окну первым обработать любые командные сообщения:

```
BOOL CMainFrame::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                          AFX_CMDHANDLERINFO* pHandlerInfo)
{
    // даем дочернему окну возможность обработать команду первым
    if (m_wndView.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

    // в противном случае выполняем обработку по умолчанию
    return CFrameWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

Пример Ex21c: MDI-приложение

Создадим MDI-программу без поддержки архитектуры «документ-вид».

1. **Средствами MFC Application Wizard создайте проект Ex21c.** На странице Application Type установите переключатель в положение Multiple documents и сбросьте флажок Document/View Architecture Support.
2. **Добавьте код прорисовки дочернего окна.** В функцию *CChildView::OnPaint* в файле ChildView.cpp добавьте выделенный код:

```
void CChildView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    dc.TextOut(0, 0, "Hello, world!");

    // Do not call CWnd::OnPaint() for painting messages
}
```

3. **Скомпилируйте и запустите приложение.** Мы получили полноценное MDI-приложение, которое никак не зависит от архитектуры «документ-вид».

Как и в Ex21b, здесь автоматически создается класс *CChildView*. Основное различие между Ex21b и Ex21c в том, что класс *CChildView*[1] создается в функции *CChildFrame::OnCreate*, а не в классе *CMainFrame*.

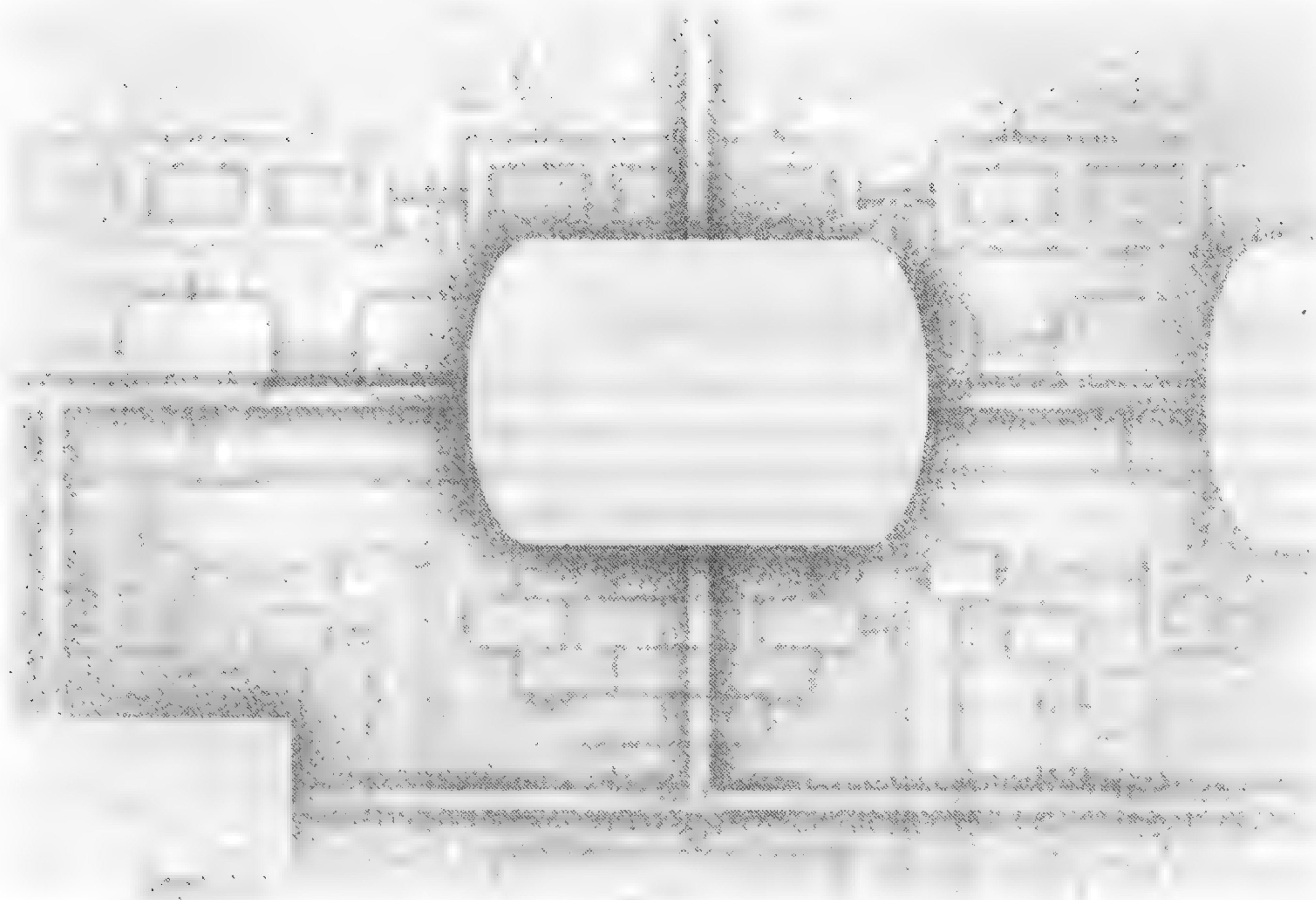
Итак, вы научились создавать приложения трех типов, не зависящие от архитектуры «документ-вид». Создание этих приложений — прекрасный способ понять работу MFC. Мы советуем вам сравнить приложения с поддержкой и без поддержки этой архитектуры, чтобы получить исчерпывающее представление о том, как классы «документ» и «вид» работают с остальной частью MFC.

[1] Точнее, объект этого класса. — *Прим. перев.*

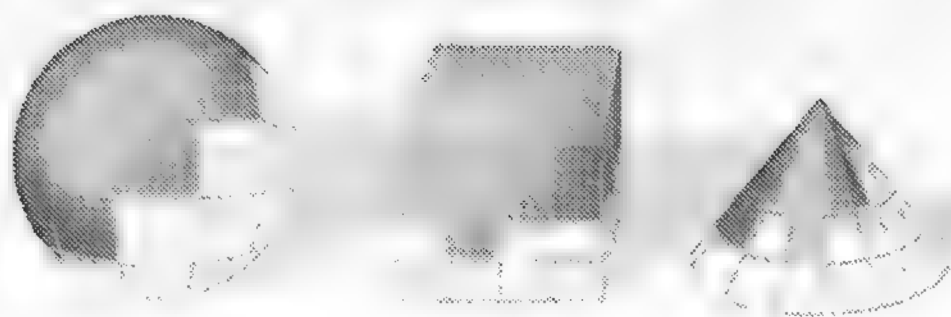
ЧАСТЬ 4

22

COM, AUTOMATION, ACTIVEX И OLE



ГЛАВА



22

Модель компонентных объектов

Модель компонентных объектов (Component Object Model, COM) лежит в основе технологии Microsoft ActiveX. Она стала неотъемлемой частью Microsoft Windows, и поэтому рассказ о ней — обязательная часть этой книги. С чего же начать? Может, с MFC-классов для элементов управления ActiveX, Automation и OLE? Но эти классы, как бы полезны они ни были, скрывают истинную архитектуру COM. Значит, начинать надо с фундаментальной теории, а она включает в себя собственно COM и интерфейсы.

В этой главе вы получите теоретические сведения, необходимые для усвоения материала следующих шести глав. Вы узнаете об интерфейсах и о том, как библиотека MFC реализует их через свои макросы и *карты интерфейсов* (interface map).

Основы технологии ActiveX

Терминология меняется столь же стремительно, как и технология, и даже внутри Microsoft нет единства относительно того, как использовать термины ActiveX и OLE. Считайте, что ActiveX — это нечто, возникшее при столкновении «старого» OLE и Интернета. ActiveX включает в себя не только те возможности Windows, основанные на COM, которые мы рассмотрим именно в этой части, но и семейство Microsoft Internet Information Server и программный интерфейс WinInet.

Да, OLE по-прежнему жива и теперь вновь расшифровывается как Object Linking and Embedding (связывание и внедрение объектов), как и в дни OLE 1.0. Сейчас это просто еще одно подмножество технологии ActiveX, содержащее всякую всячину, например операцию drag-and-drop (перетащить и отпустить). К сожалению (или к счастью, если у вас есть ранее написанный код), исходный MFC-код и Windows API не следуют за последними изменениями терминологии. Поэтому в

названиях функций и классов вы увидите множество упоминаний *OLE* и *Ole*, хотя некоторые из этих функций выходят за рамки связывания и внедрения. В этой части книги в коде, сгенерированном MFC Application Wizard, вы можете заметить упоминания о «сервере» (server). Теперь Microsoft резервирует этот термин только для серверов баз данных и Интернет-серверов. В отношении OLE-серверов применяется новый термин — *компонент* (component).

Компьютерные секции книжных магазинов забиты книгами по OLE, COM и ActiveX. Мы не обещаем достигнуть той глубины, которой отличаются эти труды, но вы наверняка получите хорошее представление о теории COM. Мы уделим большее, чем в других книгах [кроме «MFC Internals» (Addison-Wesley, 1996) Джорджа Шеферда и Скотта Уингоу (George Shepherd и Scot Wingo)], внимание связи COM с классами библиотеки MFC. Это послужит хорошей подготовкой к штудированию серьезных трудов по ActiveX/COM, в том числе «Inside OLE» (Microsoft Press, 1995) Крейга Брокшмидта (Kraig Brockschmidt) и «Essential COM» (Addison-Wesley, 1998) Дона Бокса (Don Box). Хорошая книга средней трудности — «Inside COM» (Microsoft Press, 1997) Дейла Роджерсона (Dale Rogerson).

COM приносит столько же проблем, сколько решает. Большую часть этой технологии заменит компонентная модель .NET со своими сборками (assembly) и CLR-средой (common language runtime). Тем не менее COM пока в силе. Итак, в путь.

Что такое COM

COM — это программная архитектура динамического компоновки ПО. В COM предпринята попытка решить проблемы поддержки версий (этим «грешат» DLL) и сложности механизма удаленного вызова процедур (remote procedure call, RPC).

Проблема в том, что в Windows нет стандартного способа взаимодействия между программными модулями. «Но, — скажете вы, — а как же DLL с их экспортируемыми функциями, DDE, буфер обмена и все API-интерфейсы Windows, не говоря уж об устаревших стандартах вроде VBX и OLE 1? Разве этого мало?» Увы, да. Вы не построите объектно-ориентированную операционную систему будущего из этого «зверинца» разрозненных, узкоспециализированных стандартов.

Сущность COM

У старых стандартов много недостатков. У Windows API слишком велика «площадь покрытия» — свыше 350 функций; VBX-расширения не «живут» в 32-разрядном мире; в DDE чрезмерно запутанная система *приложений* (application), *тем* (topic) и *элементов* (item); обращение к DLL всецело зависит от конкретного приложения. Модель COM предоставляет унифицированный, открытый, объектно-ориентированный протокол связи между программами, который поддерживает:

- стандартный, не зависящий от языка программирования способ загрузки и вызова Win32-модулей DLL клиентскими Win32-программами;
- универсальный способ управления одной EXE-программы другой, выполняемой на том же компьютере (замена DDE);
- элементы управления ActiveX, пришедшие на смену VBX-элементам;
- новый мощный способ взаимодействия прикладных программ с ОС;

- расширения для поддержки новых протоколов вроде интерфейса баз данных OLE DB;
- Distributed COM (DCOM) — технологию, позволяющую взаимодействовать программам на разных компьютерах, даже если процессоры этих компьютеров принадлежат к разным семействам.

Итак, что же такое COM? Задать вопрос намного проще, чем ответить на него. Главная линия, проводимая в DevelopMentor (система учебных центров разработчиков), — это «COM есть любовь». Иначе говоря, COM — это мощная интеграционная технология, позволяющая собрать разрозненные части ПО вместе в период выполнения. COM дает разработчику возможность писать интегрируемое ПО, не вдаваясь в тонкости многопоточности и не привязываясь к конкретному языку программирования.

COM — это протокол, который соединяет программные модули, а затем покидает сцену — далее модули взаимодействуют через механизм, называемый *интерфейсом* (interface). Интерфейсы не требуют статического или динамического связывания точек входа или «зашитых» в программу адресов, кроме нескольких универсальных COM-функций, активизирующих процесс установления связи. Интерфейс (точнее, COM-интерфейс) — это термин, с которым вы встретитесь еще не раз.

COM-интерфейс

Перед погружением в изучение интерфейсов давайте вспомним принципы наследования и полиморфизма в обычном C++. Для этого воспользуемся моделью межпланетных перелетов. Представьте себе космический корабль, летящий в Солнечной системе в гравитационном поле Солнца. При обычном программировании на C++ вы могли бы объявить класс *CSpaceship* и написать конструктор, задающий начальные координаты и ускорение корабля. Затем вы написали бы невиртуальную функцию-член *Fly*, которая вычисляла бы на основе законов Кеплера новые координаты корабля через определенный интервал времени, скажем, 0,1 секунды. Можно было бы написать еще и функцию *Display*, чтобы изображать в окне космический корабль. Самая интересная особенность класса *CSpaceship* — это то, что интерфейс класса C++ (т. е. протокол взаимодействия класса с клиентом) и его реализация связаны между собой очень тесно. Одна из основных целей COM как раз и состоит в отделении интерфейса класса от его реализации.

Если использовать в этом примере COM, код, моделирующий космический корабль, разместится в отдельном EXE- или DLL-файле (*компоненте*), который является COM-модулем. Программа-*клиент* не сможет вызывать *Fly* или конструктор *CSpaceship* напрямую, так как объект, описывающий космический корабль, доступен только через стандартную глобальную функцию, предоставляемую COM, в дальнейшем клиент и объект общаются через интерфейсы.

Прежде чем взяться за настоящую COM, попробуем создать ее «модель», в которой и компонент, и клиент статически скомпонованы в единый EXE-файл. Вместо упомянутой стандартной глобальной функции мы придумаем функцию *GetClassObject*. В нашей модели клиенты будут пользоваться этой глобальной абстрактной функцией (*GetClassObject*) для объектов конкретного класса. В реальной жизни

СОМ-клиенты вначале получают объект класса, а затем запрашивают у него создание реального объекта — во многом так же, как MFC выполняет динамическое создание. У *GetClassObject* три параметра:

```
BOOL GetClassObject(int nClsid, int nIid, void** ppvObj);
```

Первый параметр — *nClsid* — это 32-разрядное целое число, которое однозначно идентифицирует класс *CSpaceship*. Второй параметр, *nIid*, — уникальный идентификатор нужного нам интерфейса. Третий — указатель на интерфейс объекта. Вспомните, что мы собираемся теперь иметь дело с интерфейсами, а это не то же самое, что классы. Как выясняется, класс может иметь несколько интерфейсов, поэтому два последних параметра и обеспечивают выбор интерфейса. При благополучном завершении функция возвращает *TRUE*.

Теперь вернемся к разработке *CSpaceship*. Мы еще не говорили об интерфейсах космического корабля. СОМ-интерфейс — это базовый класс С++ (точнее, структура — *struct*), который объявляет группу чисто виртуальных функций. Эти функции полностью управляют той или иной стороной поведения производного класса. Для *CSpaceship* мы напишем интерфейс *IMotion*, который будет управлять позицией объекта — космического корабля. Простоты ради объявим только две функции: *Fly* и *GetPosition*, и пусть позиция определяется единственным целым числом. *Fly* перемещает космический корабль, а *GetPosition* возвращает его текущее положение.

```
struct IMotion
{
   virtual void Fly() = 0;
   virtual int& GetPosition() = 0;
};

class CSpaceship : public IMotion
{
protected:
   int m_nPosition;
public:
   CSpaceship() { m_nPosition = 0; }
   void Fly();
   int& GetPosition() { return m_nPosition; }
};
IMotion* pMot;
GetClassObject(CLSID_CSpaceship, IID_IMotion, (void**) &pMot);
```

Допустим пока, что СОМ с помощью уникальных идентификаторов *CLSID_CSpaceship* и *IID_IMotion* создает космический корабль, а не иной объект. В случае успеха вызова *pMot* указывает на объект *CSpaceship*, каким-то образом сконструированный функцией *GetClassObject*. Класс *CSpaceship* реализует функции *Fly* и *GetPosition*, поэтому основная программа может вызывать их для конкретного объекта — космического корабля:

```
int nPos = 50;
pMot->GetPosition() = nPos;
```

```
pMot->Fly();
nPos = pMot->GetPosition();
TRACE("new position = %d\n", nPos); // новая позиция
```

Итак, корабль стартовал и ушел в полет, мы же получили полный контроль над ним через указатель *pMot*. Заметьте: *pMot* формально не является указателем на объект *CSpaceship*, но в данном случае указатели на *CSpaceship* и на *IMotion* одинаковы, так как *CSpaceship* наследует *IMotion*. Здесь хорошо видно, как работают виртуальные функции — классический полиморфизм C++.

Усложним задачу, добавив второй интерфейс, *IVisual*, отвечающий за визуальное представление космического корабля. Для него достаточно одной функции *Display*. Готовый базовый класс выглядит так:

```
struct IVisual
{
   virtual void Display() = 0;
};
```

Вы заметили, что в COM нужно объединять функции в группы? Но зачем? В нашей модели космического пространства нам, вероятно, захочется добавить другие типы объектов — не только космические корабли. Представьте, что интерфейсы *IMotion* и *IVisual* используются другими классами. Возможно, класс Солнца, *CSun*, реализует интерфейс *IVisual*, но не *IMotion*, а класс космической станции, *CSpaceStation*, предоставляет, кроме этих двух, еще и дополнительные интерфейсы. Если вы «опубликуете» свои интерфейсы *IMotion* и *IVisual*, то не исключено, что их возьмут на вооружение другие компании, занимающиеся моделированием космического пространства.

Рассматривайте интерфейс как своего рода соглашение, контракт, заключаемый между двумя программными модулями. Идея в том, что объявления интерфейса никогда не меняются. Если понадобится обновить код, моделирующий космический корабль, интерфейсы *IMotion* и *IVisual* останутся неизменны — вы просто добавите новый интерфейс, скажем, *ICrew* — интерфейс команды корабля. При этом существующие клиентские программы продолжат работать со старыми интерфейсами, а новые смогут использовать *ICrew*. Такие клиенты способны определить в период выполнения, какие интерфейсы поддерживает данная версия ПО, моделирующего космический корабль.

Функцию *GetClassObject* можно рассматривать как более мощную альтернативу конструкторам классов и оператору *new* в C++. Последние позволяют создавать один объект с единственным набором функций-членов, а *GetClassObject* — объект и возможность «говорить» с ним (интерфейс). Чуть позже вы увидите, что работа всегда начинается с одного интерфейса, через который узнают о других интерфейсах объекта.

Так как же запрограммировать *два* интерфейса для *CSpaceship*? Можно было бы использовать характерное для C++ множественное наследование, но оно бесполезно, если два интерфейса содержат функции с одним именем. Вместо этого в библиотеке MFC применяются *вложенные классы* (nested classes). Не все программисты на C++ хорошо знакомы с этим приемом, поэтому дадим кое-какие пояснения. Вот первый фрагмент кода для класса *CSpaceship*, в котором используются вложенные интерфейсы:

```
class CSpaceship
{
protected:
   int m_nPosition;
   int m_nAcceleration;
   int m_nColor;
public:
   CSpaceship()
      { m_nPosition = m_nAcceleration = m_nColor = 0; }
   class XMotion : public IMotion
   {
   public:
      XMotion() { }
      virtual void Fly();
      virtual int& GetPosition();
   } m_xMotion;

   class XVisual : public IVisual
   {
   public:
      XVisual() { }
      virtual void Display();
   } m_xVisual;

   friend class XVisual;
   friend class XMotion;
};
```

Примечание Возможно, имеет смысл сделать *m_nAcceleration* переменной-членом класса *XMotion*, а *m_nColor* — класса *XVisual*. Мы же объявили их в классе *CSpaceship*, поскольку такая стратегия лучше совместима с MFC-макросами, в чем вы позже и убедитесь.

Заметьте: реализации *IMotion* и *IVisual* содержатся в родительском классе *CSpaceship*. В COM такой родительский класс известен как класс с идентификационными данными, или «лицом» объекта (object identity). Заметьте также, что *m_xMotion* и *m_xVisual* — внедренные переменные-члены этого класса. Можно было бы реализовать *CSpaceship* исключительно за счет внедрения. Но вложение классов дает два преимущества: во-первых, функции-члены вложенного класса получают доступ к переменным-членам родительского класса без дополнительных указателей на *CSpaceship*, и во-вторых, вложенные классы упакованы в родительском и невидимы за его пределами. Взгляните на код функции-члена *GetP*

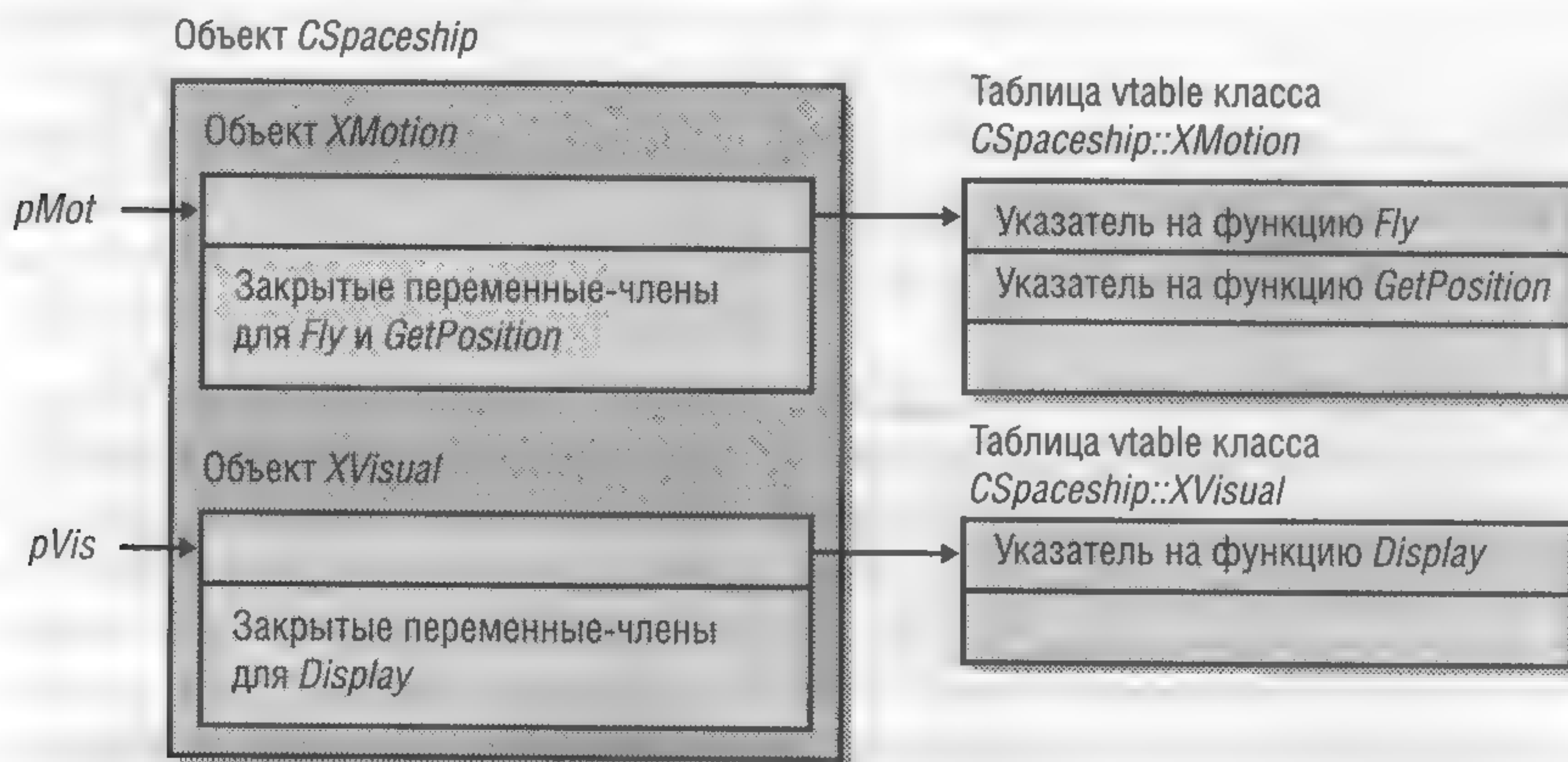
```
int& CSpaceship::XMotion::GetPosition()
{
   METHOD_PROLOGUE(CSpaceship, Motion) // создает pThis
   return pThis->m_nPosition;
}
```

Обратите внимание на оператор разрешения области видимости: он употреблен дважды, что необходимо для функций-членов вложенных классов. *METHOD_PROLOGUE* — это MFC-макрос, который с помощью стандартного оператора языка С *offsetof* генерирует *pThis* — указатель *this* для родительского класса. Компилятору всегда известно смещение от начала данных родительского класса до начала данных вложенного класса. Таким образом, функция *GetPosition* получает доступ к переменой-члену *m_nPosition* класса *CSpaceship*.

Теперь предположим, что у нас есть *два* указателя *pMot* и *pVis* для какого-то объекта *CSpaceship*. (Отложим пока вопрос о том, как их получить.) Функции-члены интерфейсов можно вызывать так:

```
pMot->Fly();
pVis->Display();
```

Что же здесь происходит «под капотом»? В С++ у каждого класса (по крайней мере в абстрактном базовом классе, у которого имеются виртуальные функции) есть *таблица виртуальных функций* (vtable). В нашем примере это означает, что такие таблицы есть у классов *CSpaceship::XVisual* и *CSpaceship::XMotion*. Для каждого *объекта* существует указатель на его данные, первый элемент которых — указатель на таблицу виртуальных функций класса. Структура указателей такова:



Примечание Теоретически COM-программы можно писать и на С. Взглянув на заголовочные файлы Windows, вы увидите код, аналогичный этому:

```
#ifdef __cplusplus
// объявления для C++
#else
/* объявления для C */
#endif
```

В С++ интерфейсы объявляются как *struct*, часто с наследованием, а в С — как *typedef struct* без наследования. В первом случае таблицы виртуальных функций ваших производных классов компилятор генерирует автоматически, тогда как при работе на С таблицы надо заполнять вручную, что весьма утомительно. Однако важно понимать, что ни в одном из языков в объявлениях интерфейсов нет ни переменных-членов, ни

конструкторов, ни деструкторов. Поэтому не полагайтесь на то, что у интерфейса есть виртуальный деструктор. (Впрочем, это не проблема — ведь деструктор для интерфейса никогда не вызывается.)

Интерфейс *IUnknown* и функция-член *QueryInterface*

Вернемся к тому, как получить указатель на интерфейс. Для этого в COM определен специальный интерфейс *IUnknown*. Фактически все интерфейсы производны от *IUnknown*, который содержит чисто виртуальную функцию-член *QueryInterface*, возвращающую указатель на интерфейс по переданному ей идентификатору интерфейса. Это предполагает, что у клиента есть указатель на какой-то интерфейс: либо на *IUnknown*, либо на производный от него. Вот новая иерархия интерфейсов, на вершине которой находится *IUnknown*:

```
struct IUnknown
{
   virtual BOOL QueryInterface(int nIid, void** ppvObj) = 0;
};
struct IMotion : public IUnknown
{
   virtual void Fly() = 0;
   virtual int& GetPosition() = 0;
};
struct IVisual : public IUnknown
{
   virtual void Display() = 0;
};
```

Чтобы выполнить требования компилятора, мы должны добавить реализацию *QueryInterface* как в *CSpaceship::XMotion*, так и в *CSpaceship::XVisual*. Как же теперь выглядят таблицы виртуальных функций? Для каждого производного класса компилятор создал таблицу, в начале которой расположены указатели на функции базового класса:

Таблица vtable класса *CSpaceship::XMotion*

Указатель на функцию *QueryInterface*
Указатель на функцию *Fly*
Указатель на функцию *GetPosition*

Таблица vtable класса *CSpaceship::XVisual*

Указатель на функцию *QueryInterface*
Указатель на функцию *Display*

Теперь функция *GetClassObject* может получить указатель на интерфейс данного объекта *CSpaceship*, получая адрес соответствующего внедренного объекта. Взглянем на функцию *QueryInterface* в классе *XMotion*:

```
BOOL CSpaceship::XMotion::QueryInterface(int nIid, void** ppvObj)
{
   METHOD_PROLOGUE(CSpaceship, Motion)
   switch (nIid) {
```

```
    case IID_IUnknown:
    case IID_IMotion:
        *ppvObj = &pThis->m_xMotion;
        break;
    case IID_IVisual:
        *ppvObj = &pThis->m_xVisual;
        break;
    default:
        *ppvObj = NULL;
        return FALSE;
    }
    return TRUE;
}
```

Поскольку *IMotion* наследует *IUnknown*, указатель на *IMotion* подойдет, даже когда вызывающая программа запросит указатель на *IUnknown*.

Примечание Стандарт COM требует: если в качестве параметра в *QueryInterface* передан *IID_IUnknown*, функция должна возвращать один и тот же указатель на *IUnknown* независимо от того, для какого указателя на интерфейс она вызвана. Поэтому, сравнив два указателя на *IUnknown*, можно определить, ссылаются ли они на один и тот же объект. *IUnknown* иногда называют «void*» для COM, потому что он представляет «лицо» (identity) объектов.

Обратимся к функции *GetClassObject*, которая на основе адреса *m_xMotion* получает первый указатель на интерфейс только что созданного объекта *CSpaceship*:

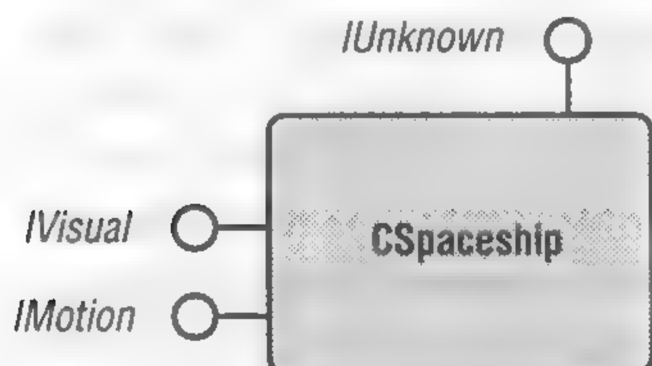
```
BOOL GetClassObject(int& nClsid, int& nIid, void** ppvObj)
{
    ASSERT(nClsid == CLSID_CSpaceship);
    CSpaceship* pObj = new CSpaceship();
    IUnknown* pUnk = &pObj->m_xMotion;
    return pUnk->QueryInterface(nIid, ppvObj);
}
```

Теперь клиентская программа может использовать *QueryInterface*, чтобы получить указатель на *IVisual*:

```
IMotion*  pMot;
IVisual* pVis;
GetClassObject(CLSID_CSpaceship, IID_IMotion, (void**) &pMot);
pMot->Fly();
pMot->QueryInterface(IID_IVisual, (void**) &pVis);
pVis->Display();
```

Заметьте: клиент, хоть и использует объект класса *CSpaceship*, *никогда* не получает указатель на сам *CSpaceship*. Таким образом, у клиента нет прямого доступа к переменным-членам *CSpaceship*, даже если бы они были открытыми. Обратите также внимание, что мы еще не пытались удалять объект *CSpaceship* — все это еще впереди.

В COM принято особое графическое представление интерфейсов и COM-классов. Интерфейсы изображаются маленькими кружками, или *гнездами* (jack), линии от которых ведут к соответствующему классу. Интерфейс *IUnknown*, поддерживаемый всеми COM-классами, изображается сверху, а остальные интерфейсы — слева от класса. Класс *CSpaceship* можно представить так:



Учет ссылок: функции *AddRef* и *Release*

В COM-интерфейсах нет виртуальных деструкторов, поэтому глупо писать:

```
delete pMot;     // pMot - указатель на IMotion;
                    // никогда так не делайте
```

В COM существует строгая процедура удаления объектов, ключевая роль в котором отводится двум функциям *IUnknown::AddRef* и *Release*. У каждого COM-класса есть переменная-член (в библиотеке MFC она называется *m_dwRef*), в которой ведется учет текущего числа «пользователей» данного объекта. Всякий раз, когда компонент возвращает указатель на новый интерфейс (скажем, при вызове *QueryInterface*), он вызывает *AddRef*, и та увеличивает счетчик *m_dwRef* на единицу. Закончив работу с указателем на интерфейс, программа-клиент, вызывает *Release*, и та уменьшает *m_dwRef* на единицу. Когда счетчик *m_dwRef* обнуляется, объект самоуничтожается[1]. Вот пример функции *Release* из класса *CSpaceship::XMotion*:

```
DWORD CSpaceship::XMotion::Release()
{
   METHOD_PROLOGUE(CSpaceship, Motion) // создает pThis
   if (pThis->m_dwRef == 0)
      return 0;
   if (-pThis->m_dwRef == 0 ) {
      delete pThis; // объект "космический корабль"
      return 0;
   }
   return pThis->m_dwRef;
}
```

В COM-программах на базе MFC конструктор объёкта устанавливает *m_dwRef* в 1, а это значит, что вызывать *AddRef* сразу после создания объекта не нужно. Тем не менее клиент должен вызвать *AddRef* при создании копии указателя на интерфейс.

---

[1] Стандарт COM не запрещает поддерживать счетчик ссылок на каждый интерфейс в отдельности, а не на весь объект в целом. — *Прим. перев.*

Фабрики класса

Терминология объектно-ориентированного программирования иногда весьма туманна. Например, программисты на Smalltalk говорят об объектах там, где программисты на C++ говорят о классах. В литературе по COM по отношению к объекту и ассоциированному с ним коду часто применяется термин *компонентный объект* (component object), а наряду с ним — *класс объекта* (class object) или *фабрика класса* (class factory). Большей точности ради его следовало бы называть *фабрикой объектов* (object factory)[1]. Объект класса в COM — это глобальная статическая область данного конкретного COM-класса. В MFC его аналогом можно считать *CRuntimeClass*. Объект класса часто называют *фабрикой класса*, так как он часто поддерживает особый COM-интерфейс — *IClassFactory*[1]. Как и другие интерфейсы, этот тоже наследует *IUnknown*. Главной функцией-членом в *IClassFactory* является *CreateInstance*, которую в нашей модели можно объявить так:

```
virtual BOOL CreateInstance(int& nIid, void** ppvObj) = 0;
```

Для чего нужна фабрика классов? Как вы уже видели, мы не в состоянии напрямую вызывать конструктор класса, но должны предоставить компоненту самому выбрать способ создания объектов. Для этого компонент предоставляет фабрику класса, инкапсулируя тем самым этап создания объекта. Поиск и запуск компонентных программных модулей для создания фабрики класса — операция «дорогая», а создание объектов посредством *CreateInstance* — «дешевая». Поэтому лучше использовать одну фабрику класса для создания множества объектов.

Что же это означает? А то, что мы все испортили, позволив функции *GetClassObject* самой создавать объекты *CSpaceship*. Следовало бы сначала создать объект — фабрику класса, а уж потом вызвать *CreateInstance*, чтобы фабрика класса (объектов) сконструировала собственно объект — космический корабль.

Сделаем все по правилам. Во-первых, объявим новый класс *CSpaceshipFactory*. Пусть для простоты он будет производным от *IClassFactory*, чтобы не связываться с вложенными классами. Кроме того, напишем код учета ссылок:

```
struct IClassFactory : public IUnknown
{
   virtual BOOL CreateInstance(int& nIid, void** ppvObj) = 0;
};
class CSpaceshipFactory : public IClassFactory
{
private:
   DWORD m_dwRef;
public:
   CSpaceshipFactory() { m_dwRef = 1; }
   // функции IUnknown
   virtual BOOL QueryInterface(int& nIid, void** ppvObj);
   virtual DWORD AddRef();
```

[1] Чтобы не путаться, можно считать, что у каждого COM-класса имеется «фабрика» для создания объектов данного класса. — *Прим. перев.*

[2] На самом деле *IClassFactory* и/или возникшие позже аналогичные ему интерфейсы поддерживаются фабрикой класса всегда. — *Прим. перев.*

```
    virtual DWORD Release();
    // функции IClassFactory
    virtual BOOL CreateInstance(int& nIid, void** ppvObj);
};
```

Затем напишем функцию-член *CreateInstance*:

```
BOOL CSpaceshipFactory::CreateInstance(int& nIid, void** ppvObj)
{
    CSpaceship* pObj = new CSpaceship();
    IUnknown* pUnk = &pObj->m_xMotion;
    return pUnk->QueryInterface(nIid, ppvObj);
}
```

И, наконец, создадим функцию *GetClassObject*, которая конструирует объект — фабрику класса и возвращает указатель на интерфейс *IClassFactory*:

```
BOOL GetClassObject(int& nClsid, int& nIid, void** ppvObj)
{
    ASSERT(nClsid == CLSID_CSpaceship);
    ASSERT((nIid == IID_IUnknown) || (nIid == IID_IClassFactory));
    CSpaceshipFactory* pObj = new CSpaceshipFactory();
    *ppvObj = pObj; // IUnknown* = IClassFactory* = CSpaceshipFactory*
}
```

Классы *CSpaceship* и *CSpaceshipFactory* работают в тандеме и совместно используют один идентификатор класса. Теперь клиентский код (без проверки ошибок) выглядит так:

```
IMotion* pMot;
IVisual* pVis;
IClassFactory* pFac;
GetClassObject(CLSID_CSpaceship, IID_IClassFactory, (void**) &pFac);
pFac->CreateInstance(IID_IMotion, &pMot);
pMot->QueryInterface(IID_IVisual, (void**) &pVis);
pMot->Fly();
pVis->Display();
```

Обратите внимание: класс *CSpaceshipFactory* реализует функции *AddRef* и *Release*. Это необходимо потому, что они — чисто виртуальные функции базового класса *IUnknown*. Мы задействуем их на следующем этапе разработки программы.

Класс *CCmdTarget*

Написанный нами код все еще далек от настоящего MFC COM, но, прежде чем перейти к реальному программированию, мы сделаем еще один небольшой шаг в развитии нашей модели COM. Как вы, наверное, догадываетесь, кое-какой код и данные можно «вынести за скобки» наших COM-классов, моделирующих космический корабль, и включить в новый базовый класс. Так и делается в библиотеке MFC, где таким базовым классом является *CCmdTarget* — стандартный базовый класс для классов документов и окон. В свою очередь *CCmdTarget* наследует классу *CObject*. Вместо настоящего класса мы используем *CSimulatedCmdTarget*, в который вынесем самый минимум — логику учета ссылок и переменную-член *m_dwRef*. Функ-

ции *ExternalAddRef* и *ExternalRelease* класса *CSimulatedCmdTarget* предназначены для вызова из производных COM-классов. В связи с использованием *CCmdTarget* мы поставим *CSpaceshipFactory* в один ряд с *CSpaceship* и реализуем интерфейс *IClassFactory* как вложенный класс.

Можно «вынести за скобки» и некоторые универсальные функции класса *CSpaceship*. Функцию *QueryInterface* можно «делегировать» вложенными классами вспомогательной функции внешнего класса *ExternalQueryInterface*, вызывающей *ExternalAddRef*. Каждая функция *QueryInterface* вызывает *AddRef*, но *CreateInstance* после *ExternalQueryInterface* вызывает функцию *ExternalRelease*. Теперь, когда функция *CreateInstance* возвратит нам первый указатель на интерфейс, счетчик ссылок на объект «космический корабль» будет равен 1. Следующий вызов *QueryInterface* увеличит счетчик до 2, и тогда для удаления объекта клиенту придется вызвать *Release* дважды.

И еще одно замечание. Мы сделаем фабрику классов глобальным объектом — тогда нам не придется вызывать ее конструктор. Когда клиент вызовет *Release*, проблем не возникнет, поскольку счетчик ссылок фабрики классов (когда клиент получает указатель на нее) равен 2. (Конструктор *CSpaceshipFactory* устанавливает счетчик в 1, а *ExternalQueryInterface*, вызванная из *GetClassObject*, увеличивает его до 2.)

Пример Ex22a: «игрушечная» COM

Листинги на следующих страницах содержат код программы Ex22a — «модели COM». Это текстовое (консольное) Win32-приложение, не использующее MFC, которое создает с помощью фабрики классов объект *CSpaceship*, вызывает функции его интерфейсов и освобождает объект. Файл Interface.h содержит объявления базового класса *CSimulatedCmdTarget* и интерфейсов, используемых как клиентом, так и компонентом. В заголовочном файле Spaceship.h хранятся объявления классов космического корабля, используемые компонентной программой. Файл Spaceship.cpp — это и есть компонент, реализующий *GetClassObject*. Client.cpp — клиент, вызывающий *GetClassObject*. Определенное «жульничество» здесь в том, что код как клиента, так и компонента скомпонован в одной исполняемой программе — Ex22a.exe. Поэтому нашей «игрушечной» COM не требуется устанавливать связь в период выполнения. (Как это делается, вы узнаете чуть позже.)

Interface.h

```
// определения, делающие наш код похожим на код MFC
#define BOOL  int
#define DWORD  unsigned int
#define TRUE   1
#define FALSE  0
#define TRACE  printf
#define ASSERT assert
//----- определения и макросы ---------------
#define CLSID_CSpaceship  10
#define IID_IUnknown      0
#define IID_IClassFactory 1
```

(*см. след. стр.*)

[illegible]

Spaceship.h

[illegible]

```
[illegible]
```

Spaceship.cpp

```
[illegible]
```

см. след. стр.

[illegible]

```
[illegible]
```

см. след. стр.

[illegible]

Client.cpp

[illegible]

```
    pMot->Fly();
    int nPos = pMot->GetPosition();
    TRACE("nPos = %d\n", nPos);
    pVis->Display();

    pClf->Release();
    pUnk->Release();
    pMot->Release();
    pVis->Release();
    return 0;
}
```

Настоящая COM с применением MFC

Поиграли, и хватит: теперь мы готовы переделать пример с космическим кораблем для истинной COM. Но сначала мы познакомимся с функцией *CoGetClassObject*, затем выясним, как COM использует реестр Windows при загрузке компонента, в чем разница между *внутренним* (in-process) (DLL) и *внешним* (out-of-process) (EXE или суррогатной DLL) компонентом и, наконец, освоим MFC-макросы, поддерживающие вложенные классы.

В результате мы должны получить DLL-компонент на базе MFC, содержащий весь код *CSpaceship*, в том числе интерфейсы *IMotion* и *IVisual*. Клиентом будет обычное MFC-приложение. Оно будет загружать компонент и работать с ним при выборе пользователем соответствующей команды в меню.

COM-функция *CoGetClassObject*

В нашей модели мы использовали фиктивную функцию *GetClassObject*. В настоящей COM применяется глобальная функция *CoGetClassObject*. [*Co* означает component object (компонентный объект).] Сравните следующий прототип с уже рассмотренной функцией *GetClassObject*:

```
STDAPI CoGetClassObject(REFCLSID rclsid, DWORD dwClsContext,
    COSERVERINFO* pServerInfo, REFIID riid, LPVOID* ppvObj);
```

Указатель на интерфейс возвращается через параметр *ppvObj*, а *pServerInfo* — это указатель на компьютер, на котором находится объект класса (*NULL*, если это локальная машина). Параметры типа *REFCLSID* и *REFIID* — ссылки на 128-разрядные *глобально уникальные идентификаторы* (globally unique identifiers, GUID) COM-классов и интерфейсов. STDAPI означает, что функция возвращает 32-разрядное значение типа *HRESULT*.

Стандартные GUID (например, GUID интерфейсов, уже созданных Microsoft) определены в Windows-библиотеках, динамически связываемых с вашей программой. Произвольные GUID, например для объектов — космических кораблей, должны определяться в программе так:

```
// {692D03A4-C689-11CE-B337-88EA36DE9E4E}
static const IID IID_IMotion = {0x692d03a4,0xc689,0x11ce,
     {0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e}};
```

Если параметр *dwClsContext* равен *CLSCTX_INPROC_SERVER*, то COM ищет DLL, а если *CLSCTX_INPROC_HANDLER* — то EXE. Эти две константы могут задаваться и одновременно (через оператор OR): для выбора либо DLL, либо EXE, исходя из соображений производительности. Например, внутренние серверы — самые быстрые, так как располагаются в адресном пространстве самой программы. Взаимодействие с EXE-серверами происходит гораздо медленнее, поскольку межпроцессные вызовы предусматривают как копирование данных, так и многочисленные переключения контекстов потоков. Возвращаемый результат — это значение типа *HRESULT*, равное 0 (*NOERROR*) при благополучном завершении функции.

Примечание Другая COM-функция — *CoCreateInstance* — объединяет в себе функциональность *CoGetClassObject* и *IClassFactory::CreateInstance*.

COM и реестр Windows

В программе Ex22a компонент и клиент статически связаны друг с другом, чего в действительности не бывает: компонентом является или DLL, или отдельный EXE-файл. Когда клиент вызывает *CoGetClassObject*, COM ищет соответствующий компонент, расположенный где-то на диске. Как же COM устанавливает связь между клиентом и компонентом? Она ищет уникальный 128-разрядный идентификатор класса в реестре. Следовательно, этот класс должен быть зарегистрирован на вашем компьютере.

Запустив редактор реестра Regedit (Regedt32 в Microsoft Windows NT), вы увидите разделы идентификаторов классов (рис. 22-1), одни из которых представляют классы, связанные с DLL (InprocServer32), а другие — с EXE (LocalServer32). Функция *CoGetClassObject* находит в реестре нужный идентификатор класса и загружает требуемый DLL- или EXE-модуль.

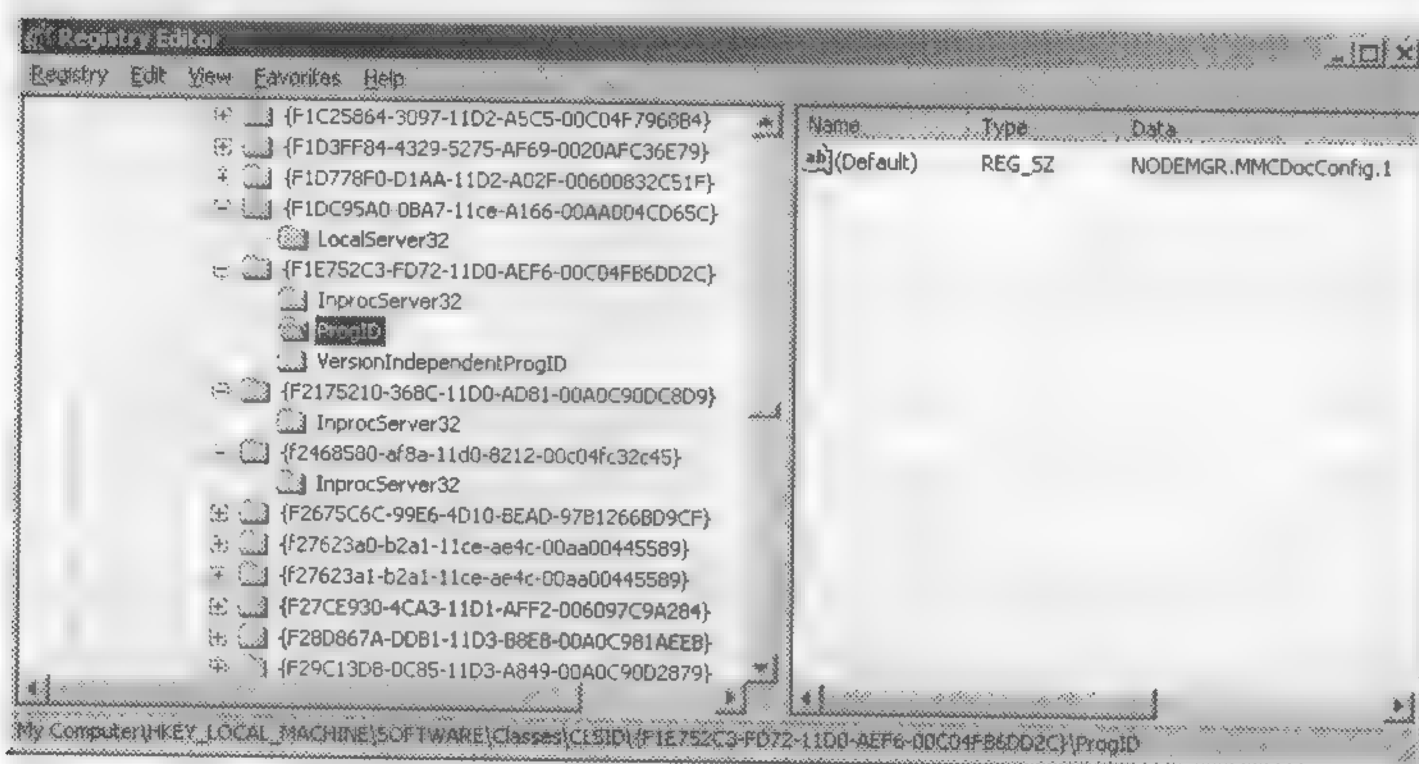


Рис. 22-1. *Разделы четырех идентификаторов классов в реестре*

Что делать, если вы не хотите отслеживать эти кошмарные идентификаторы в своей программе-клиенте? Нет проблем. COM поддерживает в своей регистрационной базе данных записи и другого типа — вполне читабельные идентификаторы программ, преобразуемые в соответствующие идентификаторы классов (рис. 22-2). Поиск в базе данных и преобразование выполняет COM-функция *CLSIDFromProgID*.

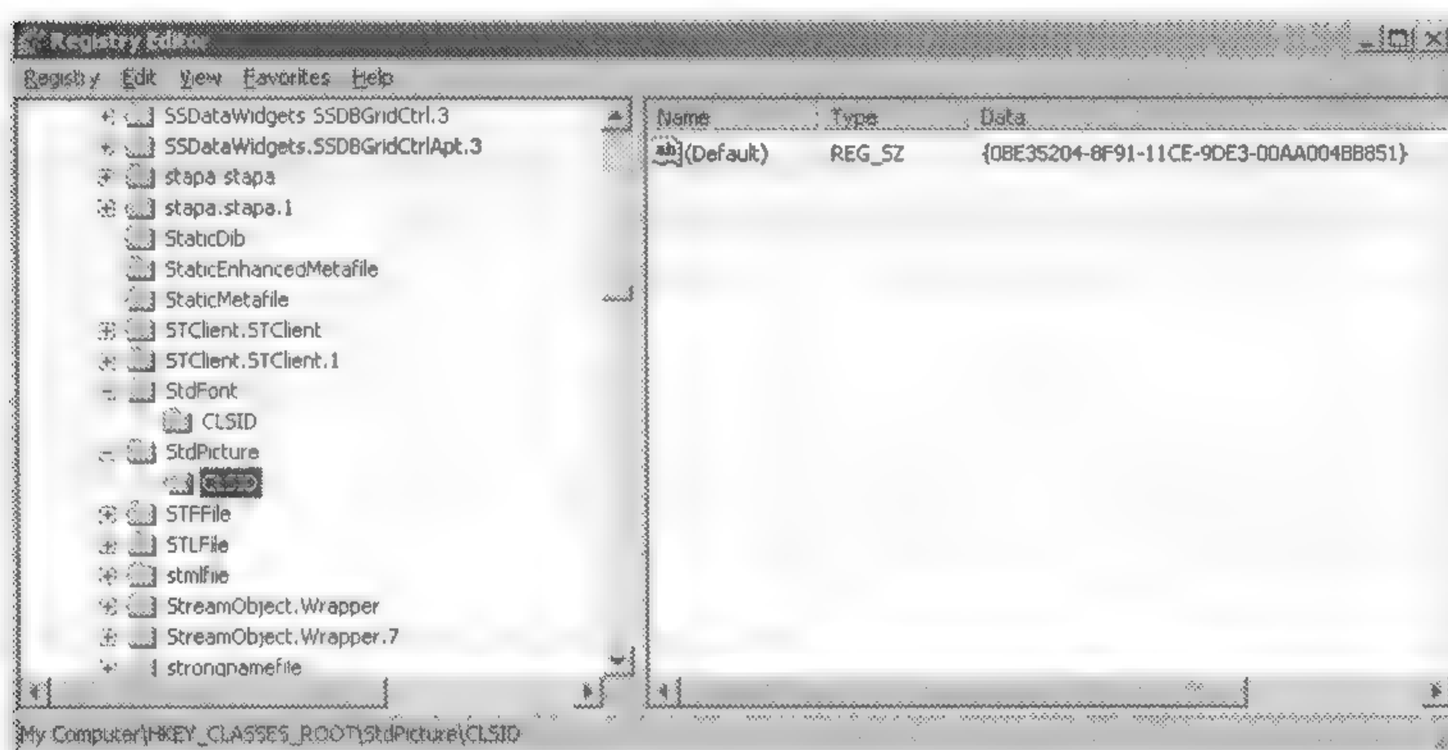


Рис. 22-2. *Читабельные идентификаторы программ в реестре*

Первый параметр функции *CLSIDFromProgID* — строка с идентификатором программы, но это необычная строка. Здесь вы впервые в COM столкнетесь с 2-байтовыми символами. Все строковые параметры COM-функций (кроме DAO) являются указателями типа *OLECHAR** на строки Unicode-символов. Теперь ваша жизнь усложнится из-за постоянного преобразования «двухбайтовых» строк в обычные и наоборот. Чтобы получить константу — строку 2-байтовых символов, ставьте перед строковым литералом символ L:

```
CLSIDFromProgID(L"Spaceship", &clsid);
```

С возможностями MFC по преобразованию Unicode-строк вы начнете знакомиться с главы 23.

Как же регистрационная информация попадает в реестр? Можно запрограммировать приложение компонента так, чтобы оно вызывало Windows-функции, напрямую модифицирующие реестр. В MFC для них предусмотрена удобная оболочка — функция *COleObjectFactory::UpdateRegistryAll*, которая находит в программе все глобальные объекты — фабрики класса и регистрирует их имена и идентификаторы классов.

Регистрация объекта в период выполнения

Вы только что видели, как реестр используется для регистрации COM-классов, хранящихся на диске. Объекты — фабрики класса тоже надо регистрировать в памяти для внешних серверов, и очень жаль, что термин «регистрировать» употребляется в обоих контекстах. Объекты в модулях внешних компонентов регистрируются в период выполнения вызовом COM-функции *CoRegisterClassObject*, и эта информация сохраняется системными DLL-модулями Windows в оперативной памяти. Если фабрика зарегистрирована в режиме, разрешающем одному экземпляру сервера создавать более одного COM-объекта, то при вызове клиентом *CoGetClassObject* COM использует существующий процесс, а не создает новый.

Вызов COM-клиентом внутреннего компонента

Мы начнем с DLL, а не с EXE-компонента, так как в этом случае взаимодействие программ проще. Приведем псевдокод, поскольку мы будем работать с MFC-классами, скрывающими многие детали.

Клиент	`CLSID clsid;` `IClassFactory* pClf;` `IUnknown* pUnk;` `CoInitialize(NULL); // инициализация COM` `CLSIDFromProgID("<имя_компонента>", &clsid);`
COM	По параметру "<*имя_компонента*>" ищет в реестре идентификатор класса
Клиент	`CoGetClassObject(clsid, CLSCTX_INPROC_SERVER, NULL,` `IID_IClassFactory, (void**) &pClf);`
COM	По идентификатору класса ищет компонент в памяти if(DLL компонента еще не загружена) { Считывает имя файла DLL из реестра Загружает DLL компонента в память процесса }
DLL компонента	if(компонент только что загружен) { Создаются глобальные объекты-фабрики классов Вызывается *InitInstance* для DLL (только в MFC) }
COM	Вызывает из DLL глобальную экспортируемую функцию DllGetClassObject со значением CLSID, переданным ранее в CoGetClassObject
DLL компонента	DllGetClassObject возвращает IClassFactory*
COM	Возвращает клиенту IClassFactory*
Клиент	`pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);`
DLLкомпонента	Вызывается функция фабрики классов CreateInstance (напрямую, через таблицу виртуальных функций компонента) Создается объект класса "имя_компонента" Возвращается указатель на запрошенный интерфейс
Клиент	`pClf->Release();` `pUnk->Release();`
DLL компонента	Через таблицу виртуальных функций вызывается Release для объекта класса "<*имя_компонента*>" if(<*счетчик_ссылок*> == 0) { Объект самоуничтожается }
Клиент	`CoFreeUnusedLibraries();`

COM	Вызывает из DLL глобальную экспортируемую функцию DllCanUnloadNow
DLL компонента	Вызвана DllCanUnloadNow `if(все созданные DLL объекты уничтожены) {` `   return TRUE;` `}`
Клиент	`CoUninitialize();   // перед самым завершением COM освобождает DLL,` `                    // если DllCanUnloadNow возвратила TRUE`
COM	Освобождает ресурсы
Клиент	Завершает работу
DLL компонента	Windows выгружает DLL, если та еще загружена и не используется другими программами

Обратите внимание на ряд особенностей. Во-первых, в ответ на вызов клиентом *CoGetClassObject* из DLL вызывается экспортируемая функция *DllGetClassObject*. Во-вторых, возвращаемый адрес интерфейса фабрики классов на деле — физический адрес[1] таблицы виртуальных функций фабрики классов в DLL. И в-третьих, клиент вызывает *CreateInstance* или *любую другую функцию интерфейса* напрямую, через таблицу виртуальных функций компонента.

Связь, которую устанавливает COM между EXE-клиентом и DLL-сервером, весьма эффективна — так же, как и вызов виртуальной функции C++ внутри процесса, плюс к этому выполняется контроль типов при компиляции. Единственная «накладка» — дополнительная операция, связанная с поиском идентификатора класса в реестре при первой загрузке DLL.

Вызов COM-клиентом внешнего компонента

Связь с отдельным EXE-компонентом через COM сложнее, чем связь с DLL-компонентом. EXE-компонент находится в другом процессе или даже на другом компьютере. Но не волнуйтесь. Пишите программы так, будто всегда есть прямая связь. О деталях позаботится COM посредством своей удаленной архитектуры, где обычно применяется RPC.

В RPC клиент обращается с вызовами к особой DLL, называемой *прокси* (proxy). Та в свою очередь передает поток данных *заглушке* (stub), которая представляет собой DLL в процессе компонента. Когда клиент вызывает функцию компонента, прокси уведомляет об этом заглушку, отправляя программе компонента сообщение, которое обрабатывается в ней скрытым окном. Механизм преобразования параметров в поток данных и обратно называется *маршалингом* (marshalling).

При использовании стандартных интерфейсов (т. е. разработанных самой Microsoft), таких как *IClassFactory* и *IPersist* [с этим интерфейсом мы познакомимся при рассмотрении постоянства (persistence) в COM], код прокси и заглушки, реализу-

---

[1] Точнее, виртуальный адрес в адресном пространстве клиентского процесса. – *Прим. перев.*

ющий маршалинг, находится в Windows-библиотеке OLEAUT32.DLL. Если же вы разрабатываете собственные интерфейсы, такие как *IMotion* и *IVisual*, то писать коды прокси и заглушки придется самостоятельно. Но к счастью, теперь для построения этого кода достаточно определить интерфейсы на специальном *языке определения интерфейсов* (Interface Definition Language, IDL) и скомпилировать код, создаваемый компилятором MIDL (Microsoft Interface Definition Language).

Теперь мы приведем псевдокод, описывающий взаимодействие EXE-клиента с EXE-компонентом. Сравните его с тем, что было показано для DLL. Обратите внимание, что вызовы со стороны клиента совершенно одинаковы.

Клиент

```
CLSID clsid;
IClassFactory* pClf;
IUnknown* pUnk;
CoInitialize(NULL); // инициализация COM
CLSIDFromProgID("<имя_компонента>", &clsid);
```

COM

```
По параметру "<имя_компонента>" ищет в реестре идентификатор класса
```

Клиент

```
CoGetClassObject(clsid, CLSCTX_INPROC_SERVER, NULL,
    IID_IClassFactory, (void**) &pClf);
```

COM

```
Ищет компонент в памяти по идентификатору класса
if(EXE-файл компонента еще не загружен или нужен, его новый экземпляр)
{
    COM считывает имя EXE-файла из реестра
    Загружает EXE компонента в память
}
```

EXE-файл компонента

```
if(только что загружен) {
    Создаются глобальные объекты-фабрики классов
    Вызывается InitInstance (только в MFC)
    CoInitialize(NULL);
    for (для каждого объекта-фабрики класса)
    {
        CoRegisterClassObject(...);
        Возвращает IClassFactory* в COM
    }
}
```

COM

```
Возвращает клиенту указатель на запрошенный интерфейс
(указатель для клиента не совпадает с указателем на интерфейс
компонента)
```

Клиент

```
pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);
```

EXE-файл компонента

```
Вызывается CreateInstance фабрики класса
(косвенно, через маршалинг)
Создается объект класса "имя_компонента"
Возвращается указатель на запрошенный интерфейс (косвенно)
```

Клиент	`pClf->Release();` `pUnk->Release();`
EXE-файл компонента	Косвенно вызывается Release для объекта класса "имя_компонента" `if(счетчик_ссылок == 0)` `{` `   Объект самоуничтожается` `}` `if(все объекты освобождены)` `{` `   Корректное завершение работы компонента` `}`
Клиент	`CoUninitialize(); // непосредственно перед завершением`
COM	Вызывает Release для всех объектов, не освобожденных клиентом
EXE-файл компонента	Завершает работу
COM	Освобождает ресурсы
Клиент	Завершает работу

Как видите, COM играет важную роль во взаимодействии клиента и компонента. COM поддерживает в памяти список фабрик классов, находящихся в активных EXE-компонентах, но не следит за отдельными COM-объектами типа *CSpaceship*. Такие объекты сами заботятся о своем уничтожении через механизм *AddRef/Release*. COM также участвует в завершении клиента. Если клиент использует EXE-сервер, COM прослушивает коммуникационный канал связи клиента с сервером, отслеживая счетчик ссылок на каждый объект. При завершении клиента COM отключается от объектов компонента, что в определенных условиях вызывает их освобождение. Но не полагайтесь на это. Перед завершением обязательно позаботьтесь, чтобы ваша клиентская программа освободила все указатели на интерфейсы.

MFC-макросы для интерфейсов

В примере Ex22a вы видели, как используются вложенные классы при реализации интерфейсов. Библиотека MFC содержит набор макросов для автоматизации этого процесса. В объявлении класса *CSpaceship*, производного от настоящего MFC-класса *CCmdTarget*, присутствуют следующие макросы:

```
BEGIN_INTERFACE_PART(Motion, IMotion)
   STDMETHOD_(void, Fly) ();
   STDMETHOD_(int&, GetPosition) ();
END_INTERFACE_PART(Motion)

BEGIN_INTERFACE_PART(Visual, IVisual)
   STDMETHOD_(void, Display) ();
```

```
END_INTERFACE_PART(Visual)

DECLARE_INTERFACE_MAP()
```

Макросы *INTERFACE_PART* порождают вложенные классы, добавляя к первому параметру префикс *X*, чтобы сформировать имя класса, и *m_x* — чтобы образовать имя внедряемого объекта. Эти макросы генерируют прототипы заданных функций интерфейса, а также прототипы *QueryInterface*, *AddRef* и *Release*.

Макрос *DECLARE_INTERFACE_MAP* порождает объявления для таблицы, содержащей идентификаторы всех интерфейсов класса. *CCmdTarget::ExternalQueryInterface* использует эту таблицу для получения указателей на интерфейсы.

В файле реализации *CSpaceship* нужны макросы:

```
BEGIN_INTERFACE_MAP(CSpaceship, CCmdTarget)
   INTERFACE_PART(CSpaceship, IID_IMotion, Motion)
   INTERFACE_PART(CSpaceship, IID_IVisual, Visual)
END_INTERFACE_MAP()
```

Они создают таблицу интерфейсов, используемую функцией *CCmdTarget::External-QueryInterface*.

Типичная функция-член интерфейса выглядит так:

```
STDMETHODIMP_(void) CSpaceship::XMotion::Fly()
{
   METHOD_PROLOGUE(CSpaceship, Motion)
   pThis->m_nPosition += 10;
   return;
}
```

Не забудьте реализовать все функции всех интерфейсов, в том числе *QueryInterface*, *AddRef* и *Release*. Последние три функции могут делегировать вызовы функциям из *CCmdTarget*.

Примечание Макросы *STDMETHOD_* и *STDMETHODIMP_* объявляют и реализуют функции, передающие параметры по правилам *__stdcall*, как того требует COM. В первом параметре этих макросов определяется тип возвращаемого значения. В двух других макросах — *STDMETHOD* и *STDMETHODIMP* — подразумевается тип *HRESULT* возвращаемого значения.

MFC-класс *COleObjectFactory*

В примере с моделью COM класс *CSpaceshipFactory* был жестко запрограммирован на генерацию объектов *CSpaceship*. В MFC применяется метод динамического создания объектов. Благодаря этому единственный класс, удачно названный *COleObjectFactory*, способен создавать объекты любого класса, указанного в период выполнения программы. Все, что от вас требуется, — вставить в объявление класса макросы, подобные этим:

```
DECLARE_DYNCREATE(CSpaceship)
DECLARE_OLECREATE(CSpaceship)
```

В файле реализации класса нужно написать макросы такого вида:

```
IMPLEMENT_DYNCREATE(CSpaceship, CCmdTarget);
// {692D03A3-C689-11CE-B337-88EA36DE9E4E}
IMPLEMENT_OLECREATE(CSpaceship, "Spaceship",0x692d03a3,0xc689,0x11ce,
   0xb3, 0x37, 0x88, 0xea, 0x36, 0xde, 0x9e, 0x4e)
```

Макросы *DYNCREATE* активизируют стандартный механизм динамического создания объектов. Макросы *OLECREATE* объявляют и определяют глобальный объект класса *COleObjectFactory* с указанным уникальным CLSID. В DLL-компоненте экспортируемая функция *DllGetClassObject*, используя глобальные переменные, определенные макросами *OLECREATE*, отыскивает нужный объект — фабрику классов и возвращает указатель на него. В EXE-компоненте инициализирующий код вызывает статическую функцию *COleObjectFactory::RegisterAll*, которая ищет все объекты-фабрики и регистрирует каждый из них вызовом *CoRegisterClassObject*. Функция *RegisterAll* вызывается и при инициализации DLL, но в этом случае она просто устанавливает флажок в объекте-фабрике (или объектах-фабриках).

Мы только коснулись проблемы поддержки COM в MFC. Подробности вы найдете к книге Шеферда и Уингоу «MFC Internals» (Addison-Wesley, 1996).

Поддержка внутренних COM-компонентов со стороны мастеров

Мастер MFC DLL Wizard не совсем подходит для создания DLL COM-компонентов, но его можно «обмануть», попросив сгенерировать обычную DLL с поддержкой Automation (для этого установите флажок Automation на странице Application Settings). В главном файле проекта представляют интерес такие функции:

```
BOOL CEx22bApp::InitInstance()
{
   CWinApp::InitInstance();
   // Register all OLE server (factories) as running.  This enables the
   //  OLE libraries to create objects from other applications.
   COleObjectFactory::RegisterAll();
   return TRUE;
}
// DllGetClassObject - Returns class factory
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   return AfxDllGetClassObject(rclsid, riid, ppv);
}
// DllCanUnloadNow - Allows COM to unload DLL
STDAPI DllCanUnloadNow(void)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   return AfxDllCanUnloadNow();
}
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
```

```
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll())
        return SELFREG_E_CLASS;

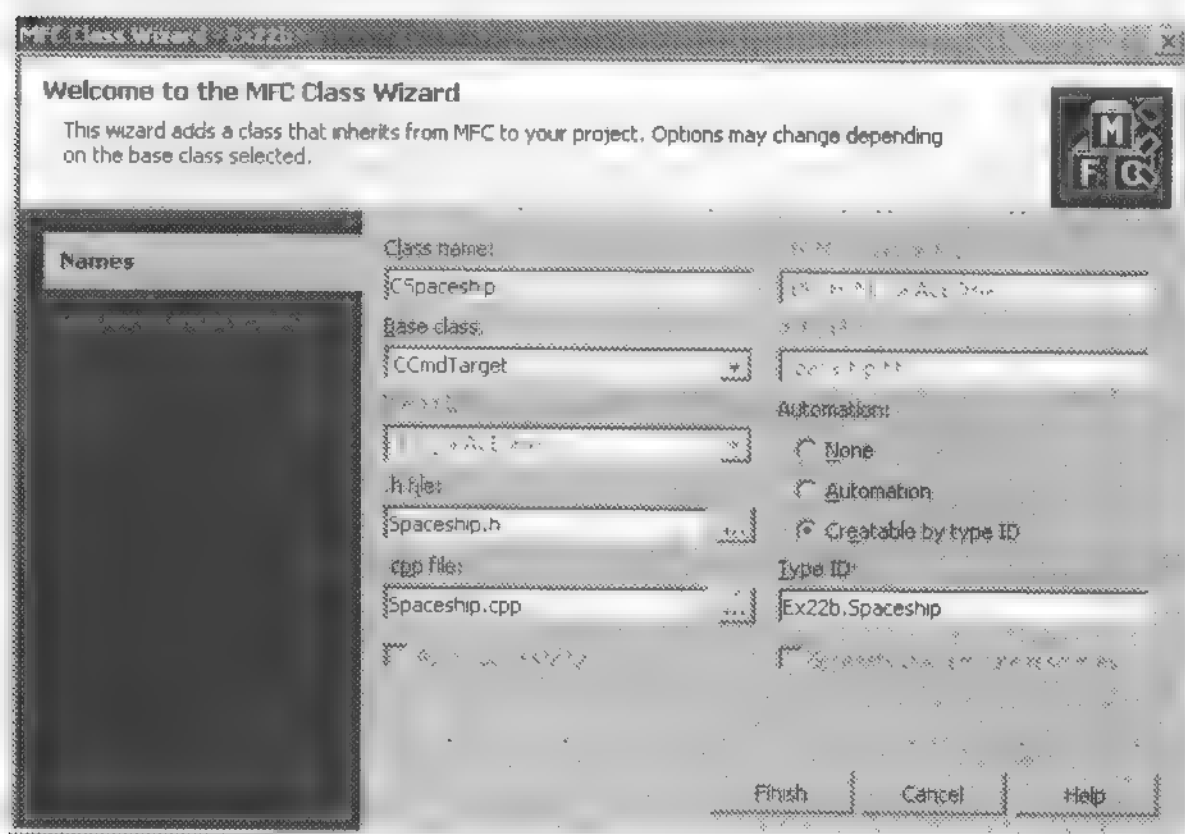
    return S_OK;
}
// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    if (!AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll(FALSE))
        return SELFREG_E_CLASS;

    return S_OK;
}
```

Эти три глобальные функции экспортируются при помощи DEF-файла проекта. Вызов MFC-функции позволяет гарантировать, что глобальные функции сделают все, что нужно DLL-модулю COM. Для обновления информации в системном реестре функции *DllRegisterServer* и *DllUnregisterServer* можно вызывать из программы-утилиты.

После создания заготовки проекта надо добавить средствами MFC Class Wizard в проект один или нескольких классов, объекты которых будут создаваться через COM. Для этого просто заполните поля имени класса, базового класса и имен файлов на странице Names:



Вновь сгенерированный класс получит ряд элементов для поддержки Automation, например, *карты диспетчеризации* (dispatch map), которые можно спокойно удалить. Кроме того, из StdAfx.h можно убрать и эти строки:

```
#include <afxodlgs.h>
#include <afxdisp.h>
```

COM-клиенты на базе MFC

Написать клиентскую COM-программу на базе MFC несложно. Вы просто создаете обычное приложение с помощью MFC Application Wizard и добавляете в StdAfx.h строку:

```
#include <afxole.h>
```

а в функцию-член *InitInstance* класса приложения — оператор:

```
AfxOleInit();
```

Теперь можете написать код, вызывающий *CoGetClassObject*.

Пример Ex22b: внутренний COM-компонент, созданный на базе MFC

Пример Ex22b — обычная MFC DLL, содержащая настоящую COM-версию класса *CSpaceship*, который вы видели в Ex22a. Файлы Ex22b.cpp и Ex22b.h сгенерированы MFC DLL Wizard, как мы только что пояснили. В файле Interface.h объявлены интерфейсы *IMotion* и *IVisual*. После текста файла Interface.h показан код класса *CSpaceship*. Сравните его с кодом из примера Ex22a. Заметили, как за счет MFC-макросов удалось уменьшить объем кода? Заметьте: учет ссылок и логику *QueryInterface* реализует MFC-класс *CCmdTarget*.

Interface.h

```
struct IMotion : public IUnknown
{
   STDMETHOD_(void, Fly) () = 0;
   STDMETHOD_(int&, GetPosition) () = 0;
};
struct IVisual : public IUnknown
{
   STDMETHOD_(void, Display) () = 0;
};
```

Spaceship.h

```
#pragma once
void ITrace(REFIID iid, const char* str);
// CSpaceship command target
class CSpaceship : public CCmdTarget
{
   DECLARE_DYNCREATE(CSpaceship)
private:
```

см. след. стр.

```
[illegible]
```

Spaceship.cpp

```
[illegible]
```

[illegible]

см. след. стр.

[illegible]

```
    TRACE("m_nColor = %d\n", pThis->m_nColor);
}
void ITrace(REFIID iid, const char* str)
{
    OLECHAR* lpszIID;
    ::StringFromIID(iid, &lpszIID);
    CString strTemp(lpszIID);
    TRACE("%s - %s\n", (const char*) strTemp, (const char*) str);
    AfxFreeTaskMem(lpszIID);
}
```

Пример Ex22c: COM-клиент на базе MFC

Ex22c — это MFC-программа, которая содержит настоящую COM-версию клиентской части примера Ex22a. Это сгенерированное MFC Application Wizard SDI-приложение с добавлением операторов *#include* для включения заголовочных файлов MFC COM, а также с добавлением вызова *AfxOleInit* для инициализации DLL. Команда Spaceship из меню Test обрабатывается функцией класса «вид», показанной в следующем листинге. В проект входит и копия файла компонента Interface.h из Ex22b. Оператор *#include* для включения этого файла помещен в начало Ex22cView.cpp.

```
void CEx22cView::OnTestSpaceship()
{
    CLSID clsid;
    LPCLASSFACTORY pClf;
    LPUNKNOWN pUnk;
    IMotion* pMot;
    IVisual* pVis;

    HRESULT hr;
    if ((hr = ::CLSIDFromProgID(L"Ex22b.Spaceship", &clsid)) != NOERROR) {
        TRACE("unable to find Program ID - error = %x\n", hr);
        return;
    }
    if ((hr = ::CoGetClassObject(clsid, CLSCTX_INPROC_SERVER,
        NULL, IID_IClassFactory, (void **) &pClf)) != NOERROR) {
        TRACE("unable to find CLSID - error = %x\n", hr);
        return;
    }

    pClf->CreateInstance(NULL, IID_IUnknown, (void**) &pUnk);
    pUnk->QueryInterface(IID_IMotion, (void**) &pMot); // должны работать
    pMot->QueryInterface(IID_IVisual, (void**) &pVis); // все 3 указателя
    TRACE("main: pUnk = %p, pMot = %p, pDis = %p\n", pUnk, pMot, pVis);

    // Тестируем все виртуальные функции интерфейсов
    pMot->Fly();
    int nPos = pMot->GetPosition();
    TRACE("nPos = %d\n", nPos);
```

```
    pVis->Display();

    pClf->Release();
    pUnk->Release();
    pMot->Release();
    pVis->Release();
    AfxMessageBox("Test succeeded. See Debug window for output.");
}
```

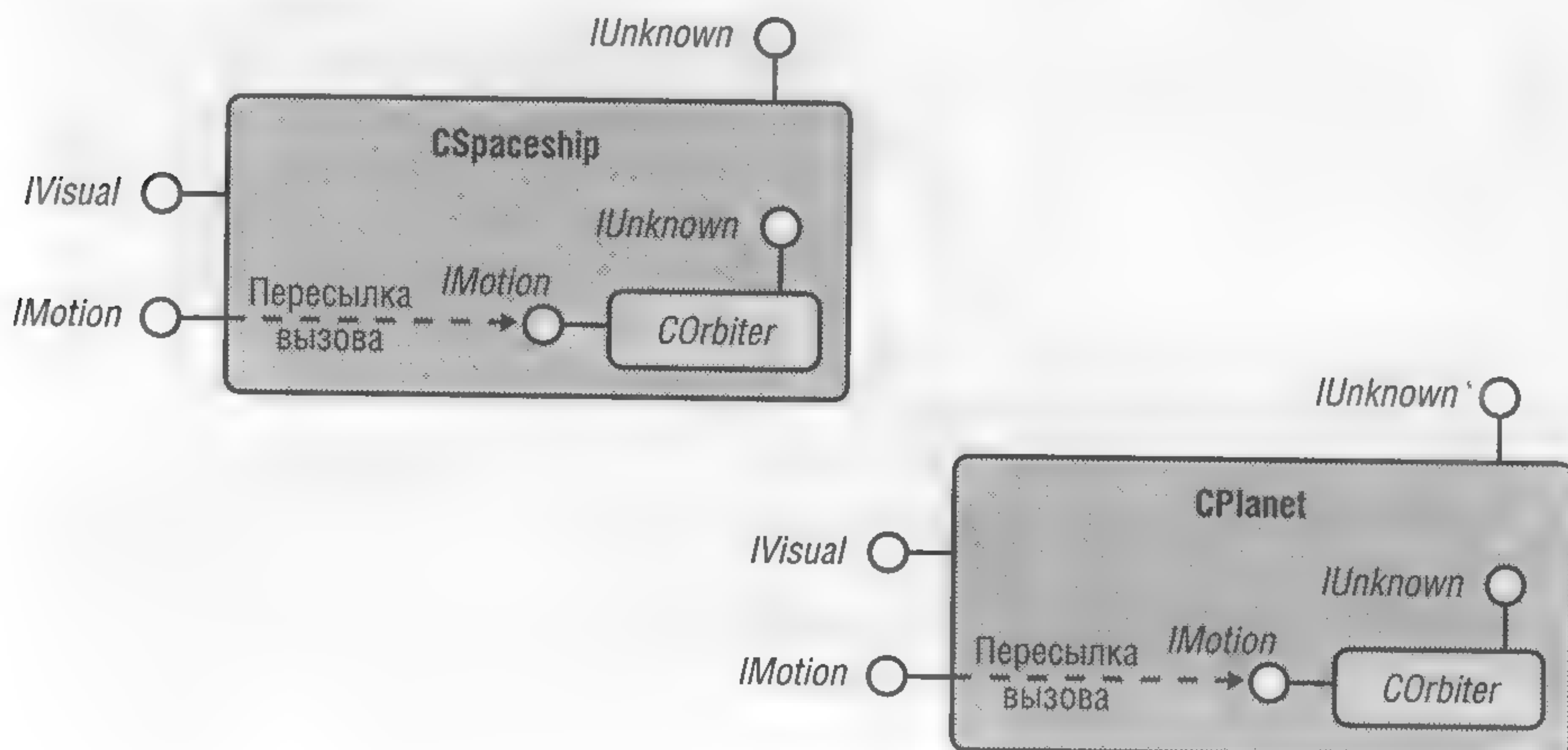
Чтобы протестировать клиент и компонент, сначала запустите компонент для обновления информации в реестре. Это можно сделать с помощью нескольких утилит, но попробуйте использовать программу REGCOMP с компакт-диска. Вам будет предложено выбрать DLL- или OCX-файл, а затем программа вызовет экспортируемую функцию *DllRegisterServer* из указанного файла.

И клиент, и компонент выводят информацию о своей работе, используя макрос *TRACE*, поэтому вам потребуется отладчик. В частности, Visual Studio .NET прекрасно подходит для запуска как клиента, так и компонента. В последнем случае надо будет ввести полное имя исполняемого файла клиента. Но копировать DLL-модуль не надо, так как Windows найдет его по информации в реестре.

Вложение, агрегирование или наследование

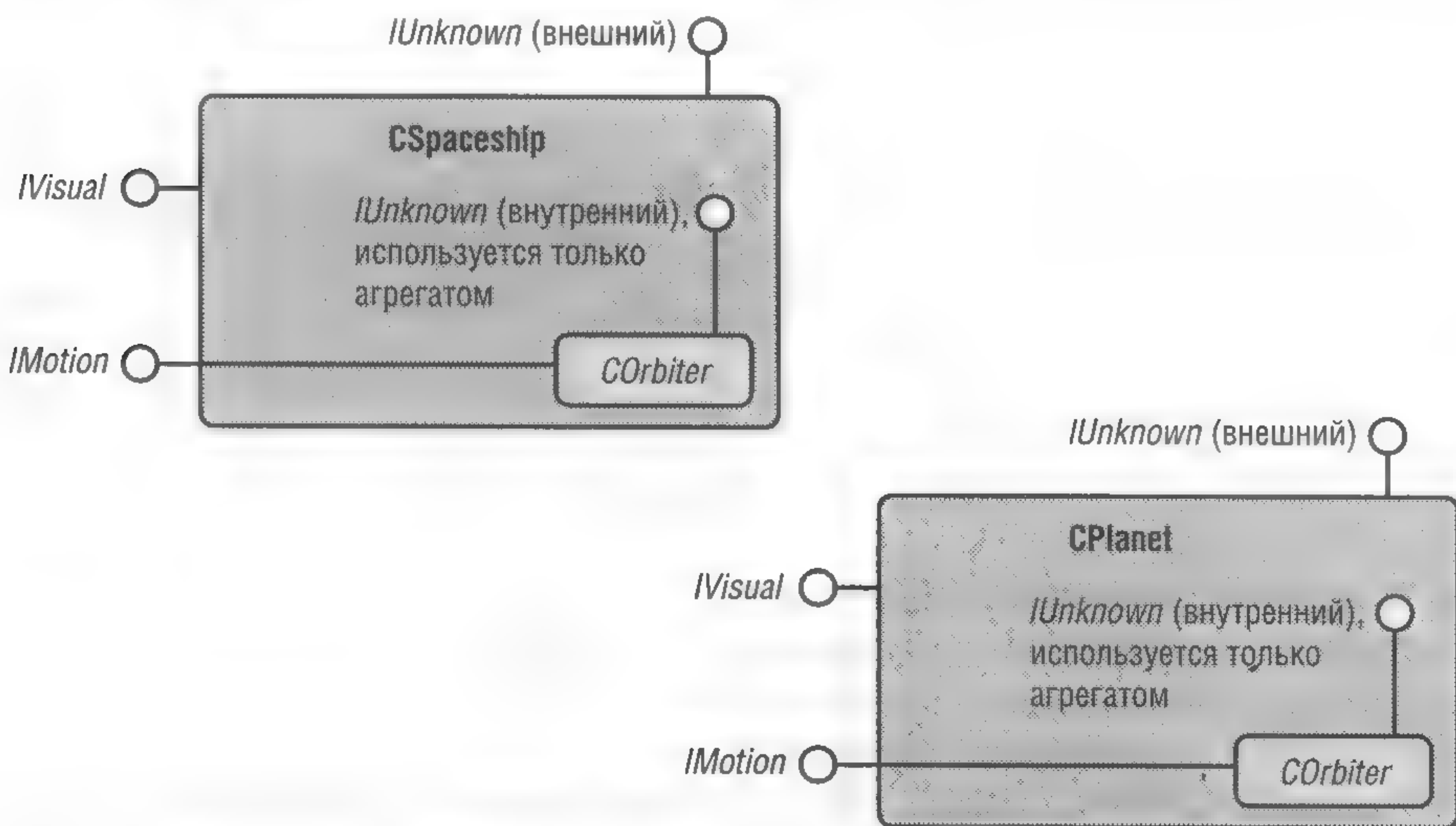
Обычно при программировании на C++ наследование применяют, чтобы вынести универсальные функции в базовый класс для повторного использования. Пример — класс *CPersistentFrame* (см. главу 14).

В COM вместо наследования используют *вложение* (containment) и *агрегирование* (aggregation). Начнем с вложения. Допустим, мы расширили нашу модель космического пространства, предусмотрев в ней движение планет. Программируя на обычном C++, мы скорее всего написали бы базовый класс *COrbiter*, который инкапсулировал бы законы небесной механики. В COM же применяются *внешние* (outer) классы *CSpaceship* и *CPlanet* и *внутренний* (inner) класс *COrbiter*. Интерфейс *IVisual* был бы реализован непосредственно во внешних классах, но интерфейсы *IMotion* они делегировали бы внутреннему классу *COrbiter*. Результат выглядел бы примерно так:



Заметьте: объект *COrbiter* не знает, что содержится внутри объекта *CSpaceship* или *CPlanet*, тогда как внешнему объекту, конечно же, известно, что в него встроен *COrbiter*. Внешний класс содержит реализации всех функций своих интерфейсов, но функции из *IMotion*, включая *QueryInterface*, просто вызывают одноименные функции внутреннего класса.

Агрегирование сложнее, чем вложение. В этом случае клиент получает прямой доступ к интерфейсам внутреннего объекта. Вот вариант модели движения планет в космосе, построенный на агрегировании:



Как и при вложении, *COrbiter* встроен (вложен) в *CSpaceship* и *CPlanet*. Допустим, клиент получил указатель на *IVisual* для *CSpaceship*, после чего вызывает *QueryInterface*, чтобы получить указатель на *IMotion*. Используя для этого указатель на внешний *IUnknown*, вы ничего не получите, так как класс *CSpaceship* не поддерживает *IMotion*. Поэтому класс *CSpaceship* получает указатель на *IMotion* от *COrbiter* с помощью указателя на внутренний *IUnknown* (встроенный объект *COrbiter*).

Теперь предположим, что у клиента есть указатель на *IMotion*, через который он вызывает *QueryInterface*, чтобы получить указатель на *IVisual*. Внутренний объект при этом должен иметь возможность передать вызов внешнему объекту, но как? Давайте приглядимся к вызову функции *CreateInstance* в листинге примера Ex22c. Первый параметр в данном случае равен *NULL*. Если вы создаете агрегированный (внутренний) объект, этот параметр применяется для передачи указателя на *IUnknown* внешнего объекта, который к тому времени уже должен существовать. Такой указатель называется ***внешним управляющим*** (controlling unknown). Класс *COrbiter* сохраняет его в одной из своих переменных-членов и использует при вызове *QueryInterface* для интерфейсов, не поддерживаемых *COrbiter*.

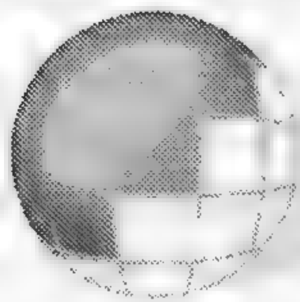
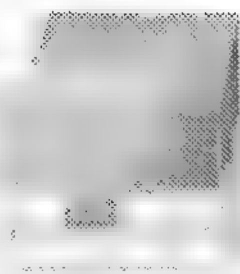
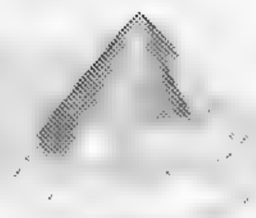
Библиотека MFC поддерживает агрегирование. У класса *CCmdTarget* есть открытая переменная-член *m_pOuterUnknown*, в которой хранится указатель на *IUnknown* внешнего объекта (если данный объект входит в состав агрегата). Функции-члены класса *CCmdTarget* — *ExternalQueryInterface*, *ExternalAddRef* и *ExternalRelease* — делегируют вызовы внешнему *IUnknown* (если он существует). Функции-члены

другой группы — *InternalQueryInterface*, *InternalAddRef* и *InternalRelease* — ничего не делегируют. Описание MFC-макросов, поддерживающих агрегирование, см. в MFC Technical Note #38.

Хотя агрегирование играет очень важную роль в COM (особенно при работе с диспетчером прокси), вам вряд ли придется использовать его в стандартных COM-приложениях.

ГЛАВА

23

Automation

Из главы 22 вы уже знаете, что такое интерфейс. Вы также познакомились с двумя стандартными COM-интерфейсами *IUnknown* и *IClassFactory*. Теперь вы готовы изучать прикладную часть COM, по крайней мере одну из ее составляющих — Automation, ранее известную как OLE Automation. В этой главе мы рассмотрим COM-интерфейс *IDispatch*, позволяющий программам на C++ взаимодействовать с программами на Visual Basic for Applications (VBA), а также с программами, написанными на других *языках сценариев* (scripting language). Кроме того, *IDispatch* играет ключевую роль при размещении COM-объектов на Web-странице. Взяв реализацию *IDispatch* из библиотеки MFC, мы напишем на C++ программы компонента и клиента Automation, причем рассмотрим как *внешние* (out-of-process), так и *внутренние* (in-process) компоненты.

Но, прежде чем программировать Automation на C++, вы должны знать, как пишет подобные программы весь остальной мир. В этой главе вы получите представление о VBA, реализованном в Microsoft Excel. Мы будем запускать из Excel свои компоненты, а также вызывать Excel из клиентской программы на C++.

Создание компонентов C++ для VBA

Отнюдь не все, кто программирует для Windows, собираются программировать на C++ и тем более углубляться в «дебри» COM. Если вы следили за событиями в отрасли последние несколько лет, то, верно, заметили тенденцию, что программисты на C++ создают модули, допускающие повторное использование, а программисты на языках более высокого уровня (Visual Basic, VBA, языки Web-сценариев и т. п.) интегрируют эти модули в приложения. Вы можете стать участником такого «разделения труда», узнав, как сделать ПО доступным для языков сценариев. Одно из средств достижения этого, поддерживаемых библиотекой MFC, — это Automation. Другое средство интеграции C++ и VBA — элементы управления ActiveX,

представляющие собой надмножество Automation, поскольку в них тоже применяется интерфейс *IDispatch*. Однако ActiveX-элементы порой просто излишни. Многие приложения, в том числе Microsoft Excel, поддерживают как компоненты Automation, так и ActiveX-элементы. Более того, все, что вы узнаете об Automation, можно применять при написании и работе с ActiveX-элементами.

Успех Automation обеспечили два обстоятельства. Во-первых, VBA (или VB Script) — сейчас стандартное средство программирования в большинстве приложений Microsoft, в том числе Word, Excel и Access, не говоря о самом Visual Basic. Все эти приложения поддерживают Automation, а значит, могут взаимодействовать с другими совместимыми с этим механизмом компонентами, в том числе с написанными на C++ и VBA. Например, можно написать на C++ программу, которая будет использовать возможности Word в обработке текста, или компонент обращения матрицы, который вызывается из VBA-макроса в таблице Excel.

Во-вторых, десятки фирм предусматривают в своих приложениях интерфейсы Automation, главным образом для программистов на VBA. Приложив минимум усилий, вы сумеете задействовать эти интерфейсы из C++, например, написать MFC-программу, управляющую программой рисования Microsoft Visio.

Automation не предназначена исключительно для программистов на C++ и VBA. Ряд фирм уже объявил о создании совместимых с Automation Basic-подобных языков, которые любой разработчик может лицензировать для включения в свое приложение, допускающее ту или иную степень программного управления. Появилась даже версия Smalltalk, поддерживающая Automation.

Клиенты и компоненты Automation

При взаимодействии на основе Automation четко проявляется иерархия типа «ведущий — ведомый» (master — slave). Ведущий — это *клиент (контроллер) Automation* [Automation client (controller)], а ведомый — *компонент (сервер) Automation* [Automation component (server)][1]. Клиент инициирует взаимодействие, создавая объект компонента (для этого может понадобиться запуск программы компонента) или подключаясь к существующему объекту в уже выполняемой программе. После этого клиент вызывает функции интерфейсов компонента и, закончив свои операции, освобождает интерфейсы.

Вот несколько сценариев такого взаимодействия.

- Automation-клиент на C++ использует в качестве компонента приложение Microsoft или сторонней фирмы. Это приводит к исполнению VBA-кода в приложении компонента.
- Automation-компонент на C++ вызывается из приложения Microsoft (или приложения на Visual Basic), которое выступает в качестве клиента. Тем самым VBA-код создает объекты C++ и оперирует с ними.
- Automation-компонент на C++ используется клиентом C++.
- Программа на Visual Basic обращается к приложению, которое поддерживает Automation, например к Excel. В этом случае Visual Basic — клиент, а Excel — компонент.

[1] Можно считать, что «клиент», «контроллер» и «контейнер» в COM — это одно и то же. — *Прим. перев.*

Microsoft Excel — лучший Visual Basic, чем сам Visual Basic

Когда были написаны первые три издания этой книги, Visual Basic мог быть клиентом Automation, но не годился для создания компонента. Но с версии 5.0 Visual Basic позволяет писать и компоненты. Поначалу мы использовали Microsoft Excel вместо VB только потому, что Excel был первым приложением Microsoft, поддерживающим синтаксис VBA, и мог выступать в роли как клиента, так и сервера. Но теперь мы используем Excel, может быть, потому, что хотим убедить программистов на C++, которые морщатся при одном упоминании о Visual Basic, приобрести Excel — хотя бы для того, чтобы вести учет своих гонораров.

Настоятельно рекомендуем приобрести копию Excel — настоящее 32-разрядное приложение из комплекта Microsoft Office. Оно позволяет писать в отдельном модуле VBA-код, осуществляющий доступ к ячейкам электронной таблицы в объектно-ориентированном стиле. В Excel легко добавлять и такие визуальные элементы, как кнопки. Словом, забудьте о старых электронных таблицах, заставлявших вас втискивать макросы в ячейки.

Эта глава — вовсе не учебник по Excel, но мы включили в нее пример с *рабочей книгой* (workbook) Excel. (Рабочая книга — файл, содержащий одну или несколько электронных таблиц и отделенный от них VBA-код.) Эта рабочая книга демонстрирует VBA-макрос, исполняемый по щелчку кнопки. Вы можете загрузить файл Demo.xls в Excel из каталога \vcppnet\Ex23a на компакт-диске или набрать его текст самостоятельно. На рис. 23-1 показана электронная таблица с кнопкой и тестовыми данными.

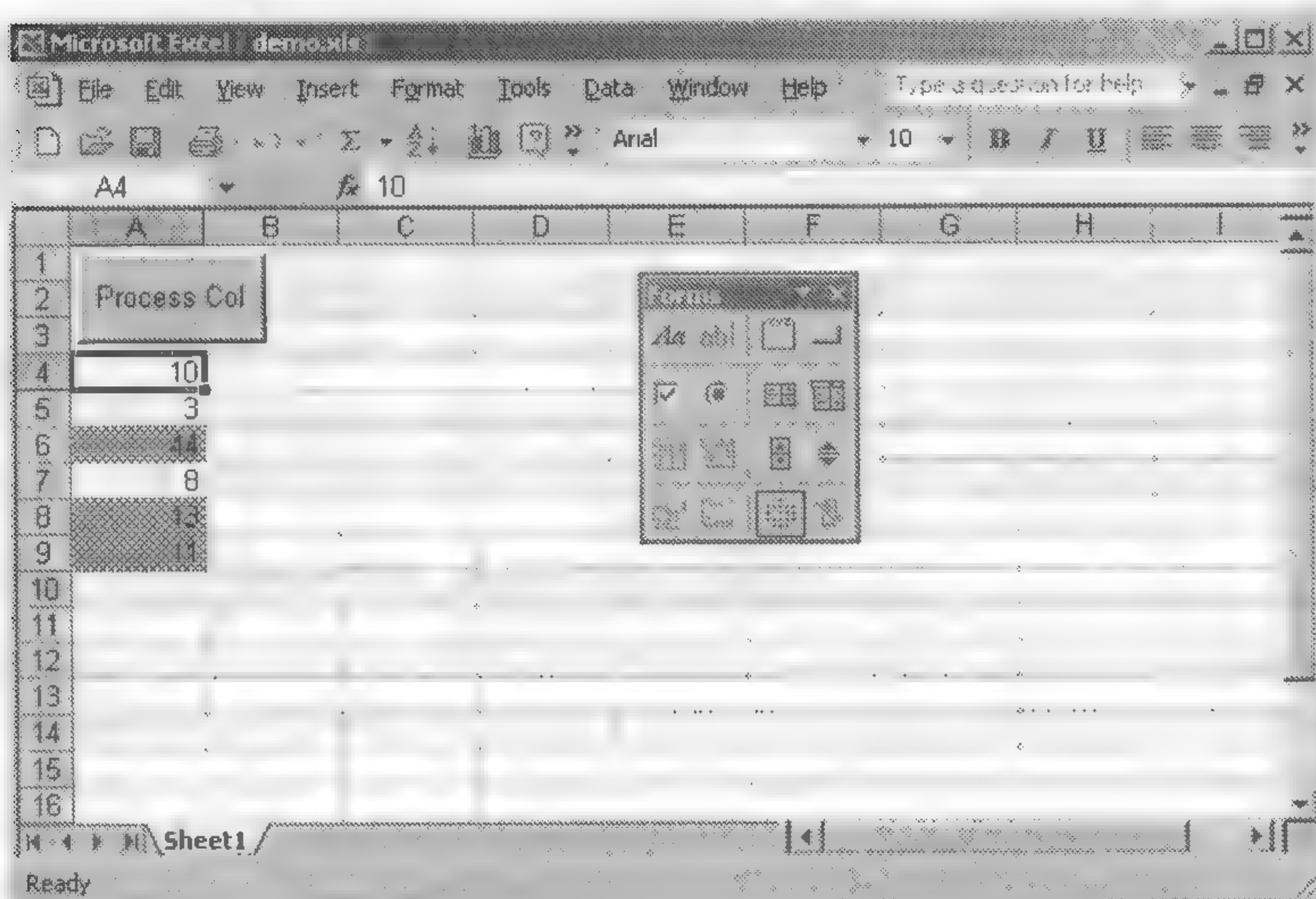


Рис. 23-1. *Электронная таблица Excel с применением VBA*

Выделите ячейки с A4 по A9 и щелкните кнопку Process Col. Программа на VBA «пройдет» по столбцу и заштрихует ячейки с числами, больше 10. На рис. 23-2 показан код макроса, который выполняет эту работу. В Excel в меню Tools (Сервис) последовательно выберите команды Macro (Макрос) и Visual Basic Editor (Редактор Visual Basic) (быстрая клавиша Alt+F11). Как видите, с этого момента мы работаем в стандартном окружении VBA.

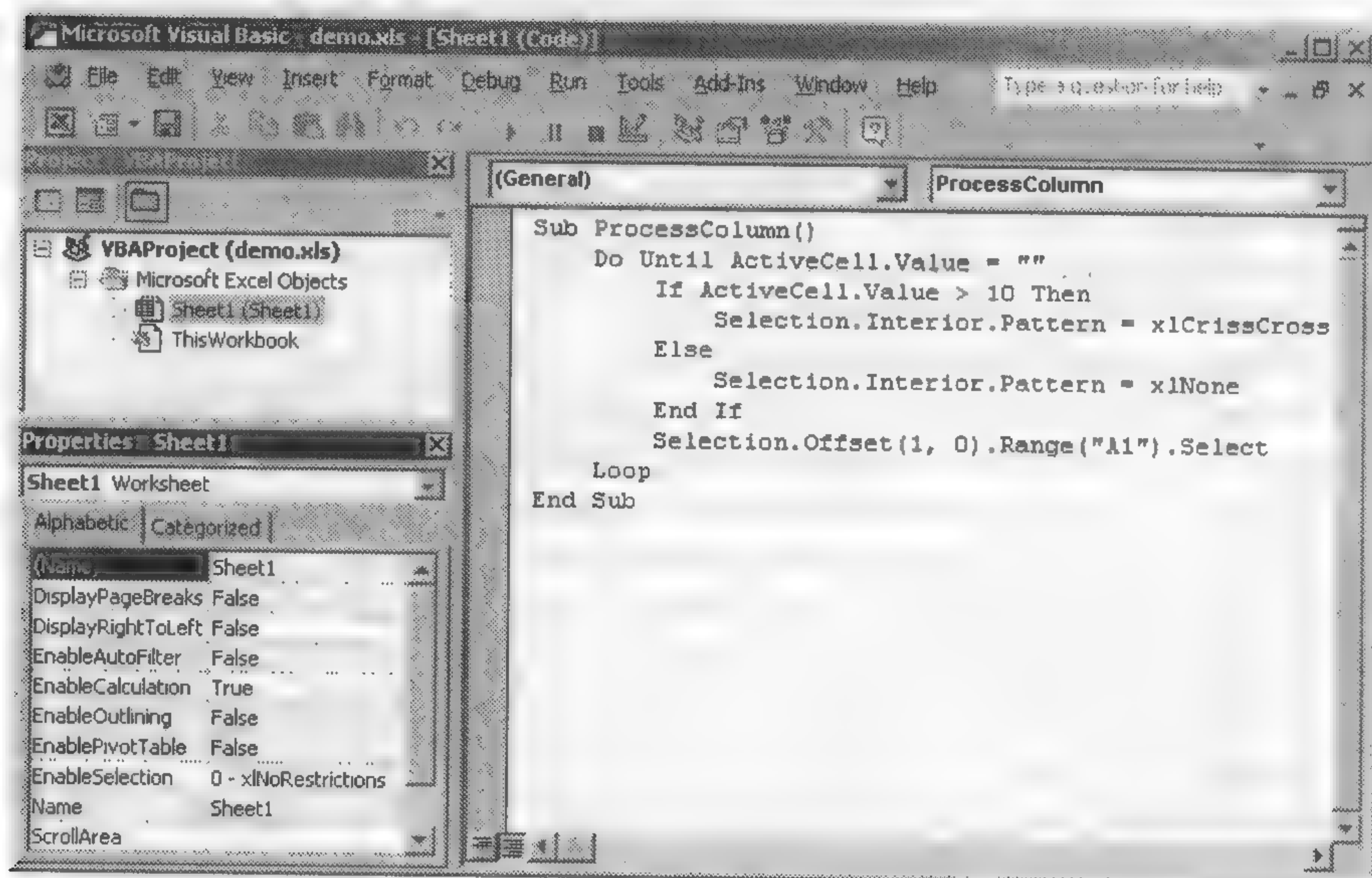


Рис. 23-2. *VBA-код для электронной таблицы Excel*

Если вы хотите подготовить этот пример самостоятельно, сделайте следующее.

1. Запустите Excel с новой рабочей книгой, нажмите Alt+F11, а затем дважды щелкните Sheet1 в левом верхнем окне.
2. Наберите код макроса, показанный на рис. 23-2.
3. Вернитесь в окно Excel, выбрав в меню File команду Close and Return to Microsoft Excel. Выберите команду Toolbar (Панели инструментов) в меню View (Вид). Для вывода на экран панели инструментов Forms (Формы) отметьте флажок Forms (Формы). (Список доступных панелей инструментов можно получить, щелкнув правой кнопкой любую из находящихся на экране инструментальных панелей.)
4. Щелкните элемент управления Button (Кнопка) и создайте кнопку, перетащив ее в левый верхний угол таблицы. Назначьте этой кнопке макрос *Sheet.Process-Column*.
5. Отрегулируйте размер кнопки и сделайте на ней надпись *Process Col* (рис. 23-1).
6. Введите любые числа в столбец, начиная с ячейки А4. Затем выделите ячейки, содержащие эти числа, и щелкните кнопку, чтобы протестировать программу.

Просто, правда?

Проанализируем один из операторов Excel VBA из нашего макроса:

```
Selection.Offset(1, 0).Range("A1").Select
```

Первый элемент (*Selection*) — это свойство не указанного явно объекта-приложения Excel. Здесь подразумевается, что его значением является объект *Range*, представляющий прямоугольный блок ячеек. Второй элемент (*Offset*) является свойством объекта *Range*, который возвращает другой объект *Range*, заданный двумя параметрами. Здесь возвращаемый объект *Range* — блок из одной ячейки, начи-

нающийся на строку ниже исходного *Range*. Третий элемент (*Range*) — это метод объекта *Range*, возвращающий еще один блок ячеек, — на этот раз левую верхнюю ячейку во втором блоке. И, наконец, метод *Select* заставляет Excel выделить указанную ячейку и сделать ее свойством Selection приложения.

По мере прохождения цикла рассматриваемый оператор смещает выделенную ячейку в таблице на следующую строку за одно повторение. К такому стилю программирования надо привыкнуть, но игнорировать его нельзя — он очень популярен в приложениях Microsoft Office, где приходится обрабатывать массу документов. Главное, всеми средствами электронной таблицы и графического ядра Excel теперь можно управлять из встроенной среды программирования.

Свойства, методы и наборы

Различие между свойством и методом в какой-то мере искусственно. По сути свойство — это некое значение, которое можно присвоить свойству и/или считать, как свойство Selection приложения Excel. Другой пример — свойство Width, характерное для многих объектов Excel. Часть свойств Excel доступна только для чтения, но большинство можно и считывать, и устанавливать.

У свойств вроде бы нет параметров, но у некоторых есть *индекс*, во многом аналогичный параметру. Он необязательно должен быть целым числом и может содержать более одного элемента (скажем, строку *и* столбец). Вы не найдете индексируемых свойств в объектной модели Excel, но Excel VBA способен работать с индексируемыми свойствами в компонентах Automation.

Методы более гибки, чем свойства. Они могут иметь несколько параметров (или вообще не иметь), присваивать или считывать данные объекта и выполнять какие-то действия, скажем, открывать окно. *Select* — пример метода, реализующего определенное действие.

Excel поддерживает *объекты-наборы* (collection objects), например объект *Sheets*, возвращаемый свойством Worksheets объекта *Application*, который содержит все электронные таблицы активной рабочей книги. Чтобы получить конкретный объект *Worksheet* из набора, можно использовать свойство Item (с целочисленным индексом) или просто применить индекс непосредственно к набору.

Интерфейсы Automation

Вы уже знаете, что COM-интерфейс — идеальный способ взаимодействия двух Windows-программ, но вы знаете и то, что разрабатывать собственные COM-интерфейсы чаще всего по крайней мере непрактично. Automation предоставляет интерфейс общего назначения *IDispatch*, удовлетворяющий программистов как на C++, так и на VBA. Наверное, вы уже догадываетесь, познакомившись с Excel VBA, что этот интерфейс основан на объектах, методах и свойствах.

Можно создавать COM-интерфейсы с функциями, имеющими параметры и возвращаемые значения любого заданного вами типа. Пример тому — *IMotion* и *IVisual* из главы 22. Но если вы собираетесь допустить к своему компоненту программистов на VBA, спешка и небрежность недопустимы. Проблему взаимодействия можно решить за счет одного интерфейса с единственной функцией-членом, достаточно «сообразительной», чтобы поддерживать методы и свойства так,

как нужно VBA. Разумеется, в *IDispatch* есть такая функция — *Invoke*. Вызов *IDispatch::Invoke* используется для COM-объектов, с которыми могут оперировать программы и на C++, и на VBA[1].

Теперь, надеемся, вы начинаете понимать задачи, решаемые Automation. Она пускает поток межмодульных взаимодействий по руслу *IDispatch::Invoke*. А как клиент впервые подключается к компоненту? Поскольку *IDispatch* — всего лишь еще один COM-интерфейс, то логика регистрации, поддерживаемая COM, применима и в этом случае. Компонентами Automation могут быть как DLL-, так и EXE-модули; доступ к ним возможен даже по сети посредством *распределенной COM* (Distributed COM, DCOM).

Интерфейс *IDispatch*

IDispatch — «сердце» Automation. Этот интерфейс, как и остальные стандартные COM-интерфейсы, полностью поддерживается за счет COM-маршалинга (т. е. реализацию его маршалинга уже написали разработчики из Microsoft) и библиотекой MFC. В компоненте должен быть COM-класс с интерфейсом *IDispatch* (и подходящая фабрика классов). Клиент получает указатель на *IDispatch* так, как это принято в COM. (Как вы увидите дальше, большую часть работы за вас будут выполнять мастера и библиотека MFC.)

Вспомните, что *Invoke* — главная функция-член *IDispatch*. В интерактивной документации Visual Studio .NET вы обнаружите у *IDispatch::Invoke* воистину кошмарный набор параметров. Но не думайте о них сейчас. Библиотека MFC действует с обеих сторон вызова *Invoke*, применяя схему вызова функций компонента на основе параметров карты диспетчеризации, которую вы определяете через макросы.

Invoke — не единственная функция *IDispatch*. Другая функция, которую может вызвать контроллер, — *GetIDsOfNames*. С точки зрения программиста на VBA свойства и методы имеют символьные имена, но программисты на C++ предпочитают более эффективные целочисленные индексы[2]. *Invoke* задает свойства и методы целыми числами, поэтому *GetIDsOfNames* очень полезна в начале программы, позволяя преобразовать все имена в индексы, если последние неизвестны при компиляции. Как вы уже видели, *IDispatch* поддерживает символьные имена для методов. Но этот интерфейс поддерживает символьные имена и для параметров методов. Эти имена функция *GetIDsOfNames* возвращает вместе с именем метода. Увы, реализация *IDispatch* в MFC не поддерживает параметры с символьными именами.

---

[1] Automation необходима для обращения к COM-объектам из языков программирования, не поддерживающих указатели на функции, поскольку по стандарту COM интерфейс — это массив указателей на функции (vtable). Согласно стандарту Automation интерфейс — это набор функций с целочисленными идентификаторами. Обращение к *IDispatch* выполняется контролером неявно, внутри транслятора или интерпретатора. — *Прим. перев.*

[2] Целочисленные индексы методов и свойств необходимы по стандарту Automation. — *Прим. перев.*

Варианты программирования в Automation

Допустим, вы собираетесь написать компонент Automation на C++. Прежде чем это сделать, нужно ответить на ряд вопросов. Каким должен быть компонент — внутренним или внешним? Какой нужен пользовательский интерфейс? И нужен ли вообще? Разрешать ли запускать EXE-компонент как автономное приложение? Если компонент будет в EXE-модуле, то в каком именно: с MDI- или с SDI-интерфейсом? Допустимо ли прямое завершение программы компонента пользователем?

Если сервер разместить в DLL, связь на основе COM окажется эффективнее, чем в случае EXE-компонента, поскольку не нужен никакой маршалинг. Однако в DLL возможности пользовательского интерфейса крайне ограниченны. Практически единственное, что доступно, — это модальное диалоговое окно. Если же вам нужен компонент, у которого есть собственное дочернее окно, понадобится ActiveX-элемент, а если к тому же вам требуется окно-рамка, придется создать внешний компонент. Как и обычная 32-разрядная DLL, DLL-библиотека Automation проецируется на адресное пространство клиентского процесса. Если два клиента обращаются к одной DLL, Windows загружает и динамически подключает ее дважды. Оба клиента при этом и не «подозревают», что работают с одним и тем же компонентом.

В случае EXE-компонента надо быть очень внимательным и различать понятия «программа компонента» и «объект компонента». Когда клиент вызывает *IClassFactory::CreateInstance* для создания объекта, тот создается фабрикой класса в компоненте, но при этом COM может как запускать, так и не запускать программу компонента.

Вот некоторые из возможных сценариев.

- **Создаваемый COM класс компонента запрограммирован так, что для создания нового объекта требуется запуск нового процесса.** В этом случае COM запускает новый экземпляр в ответ на второй и все последующие вызовы *CreateInstance*, каждый из которых возвращает указатель на *IDispatch*.
- **Особый вариант предыдущего сценария, характерный для MFC-приложений.** Класс компонента — это MFC-класс документа в SDI-приложении. Всякий раз, когда клиент вызывает *CreateInstance*, запускается новый экземпляр приложения компонента— с новыми объектами «документ», «вид» и основным окном-рамкой SDI.
- **Класс компонента допускает создание нескольких объектов в одном процессе.** Всякий раз, когда контроллер вызывает *CreateInstance*, создается новый объект компонента. Но при этом запущена только одна копия процесса компонента.
- **Особый вариант предыдущего сценария, характерный для приложений MFC.** Класс компонента — MFC-класс документа в MDI-приложении. Выполняется одна копия компонентного приложения с одним основным окном-рамкой MDI. Всякий раз, когда клиент вызывает *CreateInstance*, создаются новые объекты «документ» и «вид» вместе с новым дочерним окном-рамкой MDI.

И еще один, более интересный случай: EXE-компонент уже выполняется, когда клиент решает обратиться к существующему объекту. Это как раз относится к Excel. Когда клиенту потребуется доступ к единственному объекту-приложению

Excel, оно может быть в свернутом виде на рабочем столе. Тогда контроллер вызывает COM-функцию *GetActiveObject*, которая возвращает указатель на интерфейс уже существующего объекта. Если вызов неудачен, контроллер может создать новый объект, вызвав *CoCreateInstance*.

При уничтожении объекта компонента действуют обычные правила COM. У объектов Automation есть счетчики ссылок, и эти объекты самоуничтожаются, когда контроллер вызывает *Release*, а значение счетчика равно 0. В случае MDI-компонента, если объект Automation — MFC-документ, удаление последнего вызывает закрытие соответствующего дочернего MDI-окна. В случае SDI-компонента удаление документа приводит к завершению процесса компонента. Клиент сам отвечает за вызов *Release* для каждого интерфейса *IDispatch* перед завершением работы. Если клиент, работавший с EXE-компонентом, завершился, не освободив интерфейсы, вмешивается COM и закрывает процесс компонента. Но лучше не полагайтесь на COM, а позаботьтесь сами, чтобы ваш контроллер выполнил корректную очистку!

В общем случае клиентское приложение зачастую получает несколько указателей на интерфейсы одного объекта. Вспомним пример с космическим кораблем из главы 22, где у класса в модели COM-компонента были указатели на *IMotion* и *IVisual*. Но при использовании Automation у приложения обычно по одному указателю (*IDispatch*) на объект. Как всегда при программировании на основе COM, надо позаботиться об освобождении всех указателей на интерфейсы. В Excel, например, многие свойства возвращают указатель на *IDispatch* для новых или существующих объектов. Если вы забудете освободить указатель на интерфейс внутреннего COM-объекта, отладочная версия библиотеки MFC сообщит об утечке памяти при завершении клиентской программы.

MFC-реализация *IDispatch*

В программе компонента интерфейс *IDispatch* можно реализовать несколькими способами. Обычно эти реализации возлагают основную работу на DLL поддержки COM в Windows, вызывая COM-функцию *CreateStdDispatch* или передавая вызов *Invoke* интерфейсу *ITypeInfo*, который использует *библиотеку типов* (type library) компонента. Последняя представляет собой таблицу, адрес которой хранится в реестре и которая позволяет клиенту запрашивать у компонента символьные имена объектов, методов и свойств. Например, клиент может иметь браузер библиотеки типов, позволяющий исследовать возможности компонента.

Библиотека MFC поддерживает библиотеки типов, но не применяет их в своей реализации *IDispatch*, основанной на *карте диспетчеризации* (dispatch map). MFC-программы не вызывают *CreateStdDispatch* и не используют библиотеку типов для *IDispatch::GetIDsOfNames*, а значит, MFC не годится для создания многоязычных компонентов Automation, которые одновременно поддерживают, скажем, английские и немецкие имена свойств и методов. (*CreateStdDispatch* тоже не поддерживает многоязычные компоненты.)

Позднее вы узнаете, как клиент может применять библиотеку типов и как MFC MFC-мастера создают такие библиотеки и работают с ними. Если у вашего компонента есть библиотека типов, клиент может просматривать ее независимо от вида реализации *IDispatch*.

Компонент Automation на базе MFC

Посмотрим, что происходит в компоненте Automation (в данном случае это упрощенная версия программы-будильника Ex23c, которую нам еще предстоит обсудить). Реализация *IDispatch* в библиотеке MFC включена в базовый класс *CCmdTarget*, так что макросы *INTERFACE_MAP* вам не понадобятся. Класс компонента Automation, например *CClock*, объявляется как производный от *CCmdTarget*, а CPP-файл этого класса содержит макросы *DISPATCH_MAP*:

```
BEGIN_DISPATCH_MAP(CClock, CCmdTarget)
   DISP_PROPERTY(CClock, "Time", m_time, VT_DATE)
   DISP_PROPERTY_PARAM(CClock, "Figure", GetFigure,
                    SetFigure, VT_VARIANT, VTS_I2)
   DISP_FUNCTION(CClock, "RefreshWin", Refresh, VT_EMPTY, VTS_NONE)
   DISP_FUNCTION(CClock, "ShowWin", ShowWin, VT_BOOL, VTS_I2)
END_DISPATCH_MAP()
```

Похоже на карту сообщений MFC, да? Заголовочный файл класса *CClock* содержит связанный с макросом код:

```
public:
   DATE m_time;
   afx_msg VARIANT GetFigure(short n);
   afx_msg void SetFigure(short n, const VARIANT& vaNew);
   afx_msg void Refresh();
   afx_msg BOOL ShowWin(short n);
   DECLARE_DISPATCH_MAP()
```

Что все это значит? А то, что у класса *CClock* следующие свойства и методы.

Имя	Тип	Описание
Time	Свойство	Прямо связано с переменной-членом *m_time* этого класса.
Figure	Свойство	Индексируемое свойство, доступное через функции-члены *GetFigure* и *SetFigure*; первый параметр — индекс, второй (для *SetFigure*) — строка. (В строках хранятся цифры «XII», «III», «VI» и «IX», отображаемые на циферблате часов.)
RefreshWin	Метод	Связан с функцией-членом *Refresh*; у него нет ни параметров, ни возвращаемого значения.
ShowWin	Метод	Связан с функцией-членом *ShowWin*; параметр — типа short integer, а возвращается булево значение.

Как связаны карта диспетчеризации MFC, *IDispatch* и функция-член *Invoke*? Макросы карты диспетчеризации генерируют статические структуры данных, считываемые MFC-реализацией *Invoke*. Контроллер получает указатель на *IDispatch* для *CClock* (связь осуществляется через базовый класс *CCmdTarget*) и вызывает *Invoke*, передавая ей в качестве параметров массив указателей. MFC-реализация *Invoke*, упрятанная где-то в *CCmdTarget*, использует карту диспетчеризации *CClock* для расшифровки переданных ей указателей и либо вызывает одну из функций-членов, либо обращается к *m_time* напрямую.

Когда мы дойдем до примеров, вы увидите, что Add Class Wizard способен сгенерировать класс компонента Automation и помочь построить карты диспетчеризации.

Клиент Automation на базе MFC

Теперь перейдем к клиентской стороне Automation. Как клиент Automation, созданный на базе MFC, вызывает *Invoke*? Для этого MFC-библиотека предоставляет класс *COleDispatchDriver*. У него есть переменная-член *m_lpDispatch*, в которой хранится указатель на *IDispatch* соответствующего компонента. Чтобы оградить вас от сложной схемы передачи параметров в функцию *Invoke*, предусмотрено несколько функций-членов класса *COleDispatchDriver*, в том числе *InvokeHelper*, *GetPropery* и *SetProperty*. Каждая из них вызывает *Invoke* через указатель на *IDispatch*, связанный с компонентом и хранящийся в объекте *COleDispatchDriver*.

Допустим, в программе-клиенте есть класс *CClockDriver*, производный от *COleDispatchDriver* и управляющий объектом *CClock* в компоненте Automation. Функции, считывающие и записывающие свойство Time, выглядят так:

```
DATE CClockDriver::GetTime()
{
   DATE result;
   GetProperty(1, VT_DATE, (void*)&result);
   return result;
}

void CClockDriver::SetTime(DATE propVal)
{
   SetProperty(1, VT_DATE, propVal);
}
```

Аналогичные функции для индексируемого свойства *Figure*:

```
VARIANT CClockDriver::GetFigure(short i)
{
   VARIANT result;
   static BYTE parms[] = VTS_I2;
   InvokeHelper(2, DISPATCH_PROPERTYGET, VT_VARIANT,
            (void*)&result, parms, i);
   return result;
}
void CClockDriver::SetFigure(short i; const VARIANT& propVal)
{
   static BYTE parms[] = VTS_I2 VTS_VARIANT;
   InvokeHelper(2, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL,
            parms, i, &propVal);
}
```

И, наконец, функции для доступа к методам компонента:

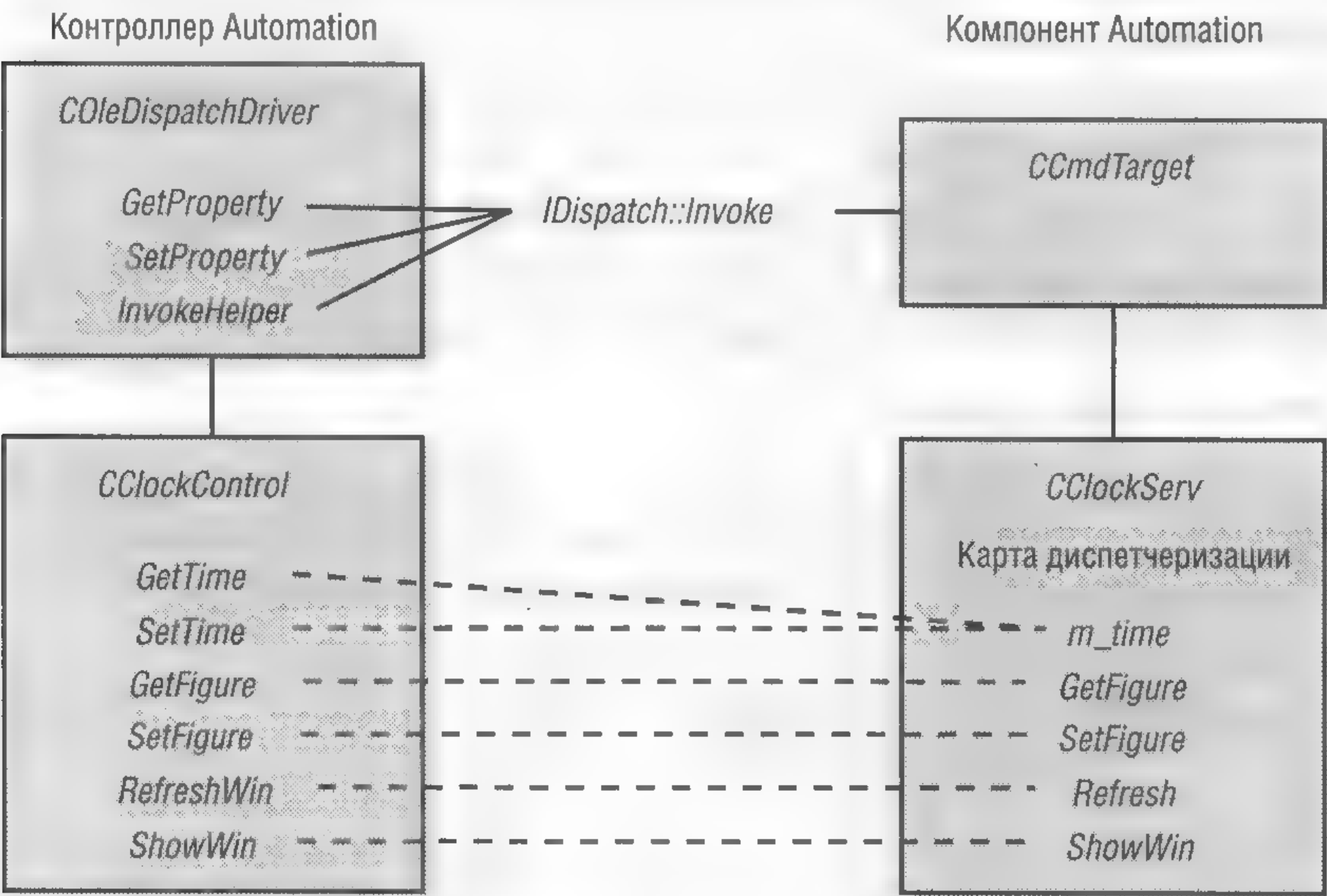
```
void CClockDriver::RefreshWin()
{
   InvokeHelper(3, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

BOOL CClockDriver::ShowWin(short i)
{
```

```
    BOOL result;
    static BYTE parms[] = VTS_I2;
    InvokeHelper(4, DISPATCH_METHOD, VT_BOOL, (void*)&result, parms, i);
    return result;
}
```

Параметры функции задают свойство или метод, его параметры и возвращаемое им значение. Чуть позже вы подробнее познакомитесь с параметрами функции диспетчеризации, а пока обратите особое внимание на первый параметр функций *InvokeHelper*, *GetPropery* и *SetProperty*. Это уникальный целочисленный индекс — *диспетчерский идентификатор* (dispatch ID, DISPID) свойства или метода. В приведенном примере эти идентификаторы можно задать при компиляции, так как компонент заранее известен. Если же вы используете сервер Automation с картой диспетчеризации, идентификаторы (индексы) определяются по порядку элементов в таблице, начиная с 1. Неизвестные диспетчерские идентификаторы компонента можно определить по символьным именам свойств или методов, вызвав функцию-член *IDispatch::GetIDsOfNames*.

Следующий рисунок иллюстрирует взаимодействие клиента (контроллера) и компонента. Сплошными линиями показаны реальные связи, осуществляемые через базовые классы MFC и функции *Invoke*, а пунктирными — логические связи между элементами классов клиента и компонента.



У большинства компонентов Automation есть файл двоичной библиотеки типов с расширением TLB. Add Class Wizard может использовать эту библиотеку типов для автоматической генерации класса производного от *COleDispatchDriver*. (Достаточно в меню Project выбрать Add Class и в отрывшемся окне — MFC Class From TypeLib.) Такой сгенерированный класс контроллера содержит функции-члены для всех методов и свойств класса с жестко заданными DISPID-идентификаторами. Иногда код, сгенерированный Add Class Wizard, требует ручного вмешательства, но лучше так, чем самому писать все функции.

После генерации класса контроллера объект этого класса встраивается в класс «вид» (или какой-либо иной класс) клиентского приложения примерно так:

```
CClockDriver m_clock;
```

Затем делается запрос к COM для создания объекта компонента часов:

```
m_clock.CreateDispatch("Ex23c.Document");
```

Теперь мы готовы к вызовам функций компонента:

```
m_clock.SetTime(COleDateTime::GetCurrentTime());
m_clock.RefreshWin();
```

Когда объект *m_clock* покидает область видимости, его деструктор освобождает указатель на *IDispatch*.

Использование директивы компиляции #import клиентом Automation

Теперь появился новый способ написания клиентских программ на основе Automation. Вместо вызова Add Class Wizard для генерации класса, производного от *COleDispatchDriver*, можно заставить компилятор создать файлы заголовков и реализации прямо из библиотеки типов. В нашем примере клиентская программа содержит инструкцию:

```
#import"..\Ex23c\debug\Ex23c.tlb" rename_namespace("ClockDriv") using namespace
ClockDriv;
```

Компилятор в этом случае создаст (и затем обработает) в подкаталоге проекта Debug или Release два файла: Ex23c.tlh и Ex23c.tli. TLH-файл содержит объявление управляющего класса *IEx23c* плюс объявление *smart-указателя* (smart pointer):

```
_COM_SMARTPTR_TYPEDEF(IEx23c, __uuidof(IDispatch));
```

Макрос *_COM_SMARTPTR_TYPEDEF* генерирует тип *IEx23cPtr*, инкапсулирующий указатель на *IDispatch* компонента. TLI-файл содержит *встраиваемую* (inline) реализацию функций-членов, несколько примеров которых приведены ниже:

```
inline HRESULT IEx23c::RefreshWin ( ) {
   return _com_dispatch_method(this, 0x4, DISPATCH_METHOD,
                         VT_EMPTY, NULL, NULL);
}
inline DATE IEx23c::GetTime ( ) {
   DATE _result;
   _com_dispatch_propget(this, 0x1, VT_DATE, (void*)&_result);
   return _result;
}
inline void IEx23c::PutTime ( DATE _val ) {
   _com_dispatch_propput(this, 0x1, VT_DATE, _val);
}
```

Обратите внимание на сходство этих функций с функциями *COleDispatchDriver*, которые мы уже рассматривали. Функции *_com_dispatch_method*, *_com_dispatch_propget* и *_com_dispatch_propput* находятся в библиотеке периода выполнения.

В клиентской программе переменная-член объекта smart-указателя встраивается в класс «вид» (или иной класс) приблизительно так:

```
IEx23cPtr m_clock;
```

затем создается объект компонента при помощи оператора:

```
m_clock.CreateInstance(__uuidof(Document));
```

Теперь для вызова функций-членов, определенных в TLI-файле, можно применить перегруженный оператор -> класса *IEx23cPtr*:

```
m_clock->PutTime(COleDateTime::GetCurrentTime());
m_clock->RefreshWin();
```

Когда объект smart-указателя *m_clock* покидает область видимости, его деструктор вызывает COM-функцию *Release*.

Директива #import — будущее COM-программирования. С каждой новой версией Visual C++ возможности COM «переползают» в компилятор, впрочем, это же верно по отношению к архитектуре «документ-вид».

Тип данных *VARIANT*

Несомненно, вы обратили внимание на тип *VARIANT* в коде клиента и компонента из предыдущего примера. *VARIANT* — это универсальный тип данных, используемый *IDispatch::Invoke* для пересылки параметров и возвращаемых значений. Он очень хорошо подходит для обмена данными с VBA. Взгляните на упрощенную версию определения типа *VARIANT* в заголовочных файлах Windows:

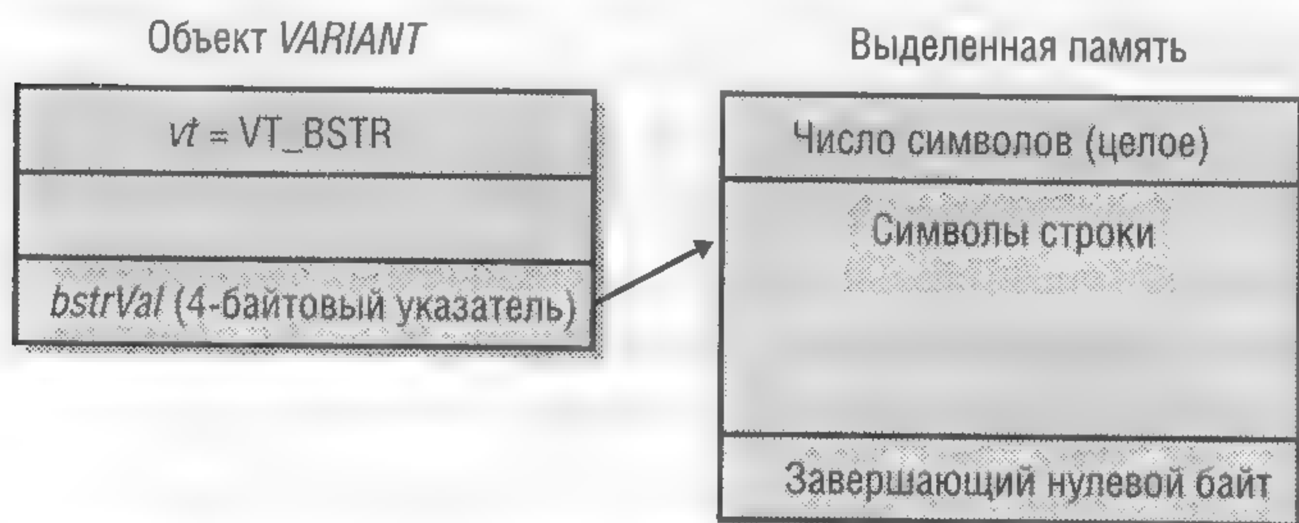
```
struct tagVARIANT {
    VARTYPE vt;  // код типа беззнаковое короткое целое
    WORD wReserved1, wReserved2, wReserved3;
    union {
        short       iVal;           // VT_I2 короткое целое
        long        lVal;           // VT_I4 длинное целое
        float       fltVal;         // VT_R4 4-байтовое с плавающей запятой
        double      dblVal;         // VT_R8 8-байтовое с плавающей запятой
        DATE        date;           // VT_DATE хранится как число
                                    // типа double в формате "дата.время"
        CY          vtCY            // VT_CY 64-разрядное целое
        BSTR        bstrVal;        // VT_BSTR
        IUnknown*   punkVal;        // VT_UNKNOWN
        IDispatch*  pdispVal;       // VT_DISPATCH
        short*      piVal;          // VT_BYREF | VT_I2
        long*       plVal;          // VT_BYREF | VT_I4
        float*      pfltVal;        // VT_BYREF | VT_R4
        double*     pdblVal;        // VT_BYREF | VT_R8
        DATE*       pdate;          // VT_BYREF | VT_DATE
        CY*         pvtCY;          // VT_BYREF | VT_CY
        BSTR*       pbstrVal;       // VT_BYREF | VT_BSTR
    }
};

typedef struct tagVARIANT VARIANT;
```

Как видите, *VARIANT* — это структура языка С, содержащая код типа *vt*, несколько байтов-заполнителей и большое *объединение* (union) уже известных вам типов. Если значение *vt* равно, скажем, VT_I2, значение *VARIANT* следует считывать из поля *iVal*, содержащего двухбайтовое целое. Если же *vt* равно VT_R8, используется поле *dblVal*, где хранится 8-байтовое число.

Объект *VARIANT* может содержать как сами данные, так и указатель на них. Если в поле *vt* установлен бит *VT_BYREF*, для доступа к данным следует применить указатели *piVal, plVal* и т. д. *VARIANT* может содержать указатель на *IUnknown* или *IDispatch*. Это значит, что в вызове Automation можно передать целый COM-объект, но если вы хотите, чтобы этот объект обрабатывался в VBA, класс объекта должен поддерживать интерфейс *IDispatch*.

Строки — особый случай. Тип *BSTR* — еще один способ представления строк символов. Переменная этого типа является указателем на массив символов, заканчивающийся нулевым символом[1]. Количество символов в массиве хранится непосредственно перед его началом. Так что переменная *BSTR* может содержать неотображаемые (бинарные) символы, в том числе нули. Если имеется объект *VARIANT* с *vt* = VT_BSTR, содержимое памяти выглядит так:



Так как строка заканчивается нулевым символом, то с *bstrVal* можно работать, как с обычным указателем на *char*, но при этом следует быть исключительно внимательным к освобождению памяти. Этот указатель нельзя просто удалить, поскольку выделенная область памяти начинается с числа символов. Для размещения и удаления объектов *BSTR* в OLE предусмотрены функции *SysAllocString* и *SysFreeString*.

Примечание *SysAllocString* — еще одна COM-функция, параметром которой служит строка 2-байтовых символов. Это означает, что все *BSTR*-строки содержат 2-байтовые символы, даже если вы не определили макрос *_UNICODE. Будьте внимательны*[2].

Для работы с *VARIANT* в Windows есть несколько полезных функций, в том числе приведенные в таблице. Если *VARIANT* содержит *BSTR*, эти функции гарантируют правильное выделение и освобождение памяти. Функции *VariantInit* и *VariantClear*

---

[1] То есть с кодом, равным 0. — *Прим. перев.*

[2] Так как все *BSTR*-строки содержат символы UNICODE, работать с такими строками, как с указателями на *char*, нельзя вопреки утверждению автора. Из этого же следует, что в конце строки находятся два нулевых байта, а не один, как показано на рисунке. — *Прим. перев.*

присваивают *vt* значение *VT_EMPTY*. Все функции для *VARIANT* — глобальные и в качестве параметра принимают значение типа *VARIANT**.

Функция	Описание
VariantInit	Инициализирует VARIANT.
VariantClear	Очищает VARIANT.
VariantCopy	Очищает память, связанную с VARIANT-получателем, и копирует в него VARIANT-источник.
VariantCopyInd	Очищает память, связанную с VARIANT-получателем, и производит преобразования, чтобы скопировать VARIANT без флага VT_BYREF.
VariantChangeType	Изменяет тип VARIANT-значения.

Класс *COleVariant*

Есть смысл написать класс-оболочку C++ структуры *VARIANT*. Конструктор класса вызывал бы *VariantInit*, а деструктор — *VariantClear*. У такого класса можно предусмотреть конструктор для каждого стандартного типа, а также конструктор копии и оператор присваивания, которые вызывали бы *VariantCopy*. При передаче управления за пределы видимости блока, в котором объявлен объект этого класса, происходил бы вызов деструктора, и память освобождалась бы автоматически.

Именно такой класс уже написан разработчиками MFC. Он отлично работает и с клиентами и компонентами Automation. Вот упрощенное объявление класса:

```
class COleVariant : public tagVARIANT
{
// Конструкторы
public:
    COleVariant();

    COleVariant(const VARIANT& varSrc);
    COleVariant(const COleVariant& varSrc);

    COleVariant(LPCTSTR lpszSrc);
    COleVariant(CString& strSrc);

    COleVariant(BYTE nSrc);
    COleVariant(short nSrc, VARTYPE vtSrc = VT_I2);
    COleVariant(long nSrc, VARTYPE vtSrc = VT_I4);

    COleVariant(float fltSrc);
    COleVariant(double dblSrc);
    COleVariant(const COleDateTime& dateSrc);
// Деструктор
    ~COleVariant();     // освобождает BSTR
// Операции
public:
    void Clear();       // освобождает BSTR
    VARIANT Detach();   // подробности позже
    void ChangeType(VARTYPE vartype, LPVARIANT pScr = NULL);
};
```

Кроме того, в классах *CArchive* и *CDumpContext* предусмотрены операции сравнения, присваивания, приведения типов и дружественные операции вставки/извлечения. Полное описание этого MFC-класса *COleVariant* см. в интерактивной документации.

А теперь посмотрим, как класс *COleVariant* помогает писать функцию компонента *GetFigure*, ссылку на которую вы видели в приведенной выше карте диспетчеризации. Пусть компонент хранит строки для четырех цифр в переменной-члене:

```
private:
   CString m_strFigure[4];
```

Вот что надо сделать при прямом использовании структуры *VARIANT*:

```
VARIANT CClock::GetFigure(short n)
{
   VARIANT vaResult;
   ::VariantInit(&vaResult);
   vaResult.vt = VT_BSTR;
   // CString::AllocSysString создает BSTR
   vaResult.bstrVal = m_strFigure[n].AllocSysString();
   return vaResult;    // Копируем vaResult,. не копируя BSTR
                       // BSTR надо будет освободить позднее
}
```

А вот то же самое, но с использованием *COleVariant*:

```
VARIANT CClock::GetFigure(short n)
{
   return COleVariant(m_strFigure[n]).Detach();
}
```

Вызов функции *COleVariant::Detach* здесь совершенно необходим. Функция *GetFigure* создает временный объект, содержащий указатель на *BSTR*. Этот объект побитно копируется в возвращаемое значение. Если вы не вызовете *Detach*, деструктор *COleVariant* освободит память, занимаемую *BSTR*, и вызывающий процесс получит *VARIANT* с указателем, ссылающимся «в никуда», точнее, просто на какой-то «мусор».

VARIANT-параметры функций компонента, вызываемых через *IDispatch*, объявлены как *const VARIANT&*. Внутри функции тип указателя на *VARIANT* всегда можно преобразовать в указатель на *COleVariant*. Вот пример функции *SetFigure*:

```
void CClockServ::SetFigure(short n, const VARIANT& vaNew)
{
   COleVariant vaTemp;
   vaTemp.ChangeType(VT_BSTR, (COleVariant*) &vaNew);
   m_strFigure[n] = vaTemp.bstrVal;
}
```

Примечание Помните, что все *BSTR*-строки содержат 2-байтовые символы. В классе *CString* есть конструктор и оператор присваивания для типа *LPCWSTR* (указатель на 2-байтовые символы). Таким образом, строка

m_strFigure будет содержать однобайтовые символы, хотя *bstrVal* и указывает на массив 2-байтовых символов.

У *VARIANT*-параметров функций *клиента*, вызываемых через *IDispatch*, тот же тип — *const VARIANT&*. Вы можете передавать в них как *VARIANT*, так и объект *COleVariant*. Взгляните на такой вызов функции *CClock::SetFigure*:

```
pClockDriver->SetFigure(0, COleVariant("XII"));
```

Примечание Вы вправе также задействовать для *BSTR* и *VARIANT* стандартные, независимые от библиотеки MFC классы *_bstr_t* и *_variant_t*. Первый инкапсулирует тип *BSTR*, второй — *VARIANT*. Оба класса корректно выделяют и освобождают память. Подробнее об этих классах см. документацию по Visual Studio .NET.

Преобразования типов параметров и возвращаемых значений для *Invoke*

Все параметры и возвращаемые значения *IDispatch::Invoke* обрабатываются в ней как данные типа *VARIANT*. Помните об этом! Реализация *Invoke* в библиотеке MFC способна сама выполнять преобразования между *VARIANT* и любым переданным вами типом (если такое преобразование возможно), что обеспечивает определенную гибкость в объявлении типов параметров и возвращаемых значений. Допустим, контроллерная функция *GetFigure* возвращает тип *BSTR*. Если компонент возвращает *int* или *long*, все в порядке: OLE и библиотека MFC преобразуют число в строку. А если компонент принимает параметр *long*, тогда как контроллер передает *int*? И вновь никаких проблем.

Примечание MFC-клиент Automation задает ожидаемый тип возвращаемого значения как параметр *VT_* функций *GetProperty*, *SetProperty* и *InvokeHelper* класса *COleDispatchDriver*. MFC-компонент Automation указывает ожидаемые типы параметров как параметры *VTS_* в макросах *DISP_PROPERTY* и *DISP_FUNCTION*.

В отличие от C++ в VBA нет строгого контроля типов. Переменные в VBA внутренне часто представляются как *VARIANT*. Возьмем, например, значение в ячейке электронной таблицы Excel. Пользователь может ввести в ячейку текстовую строку, целое значение, число с плавающей запятой или дату. VBA рассматривает данные в ячейке как *VARIANT* и возвращает клиентам Automation объект именно этого типа. Если возвращаемое значение функции клиента объявлено как *VARIANT*, контроллер может проверить значение *vt* и соответствующим образом обработать данные.

В VBA применяется формат даты/времени, отличный от формата MFC-класса *CTime*. Переменные типа *DATE* хранят и дату, и время как единое значение типа *double*. Дробная часть представляет время (0,25 соответствует 6 часам утра), а целая — дату (число дней, прошедших с 30 декабря 1899 г.)[1]. Библиотека MFC пре-

[1] Отрицательные значения соответствуют датам до 30 декабря 1899 г. — *Прим. перев.*

доставляет класс *COleDateTime*, облегчающий работу с датами. Дату вы можете сформировать так:

```
COleDateTime date(2001, 2, 11, 18, 0, 0); // 11 февраля 2001 г., 6 часов вечера
```

В классе *COleVariant* определен оператор присваивания для *COleDateTime*, а у *COleDateTime* есть функции-члены для выделения компонентов времени и даты. Вот как можно вывести показания времени:

```
TRACE("time = %d:%d:%d\n",
    date.GetHour(), date.GetMinute(), date.GetSecond());
```

Если у вас есть переменная *VARIANT*, содержащая значение типа *DATE*, то для преобразования даты в строку можно вызвать функцию *COleVariant::ChangeType*:

```
COleVariant vaTimeDate = date;
COleVariant vaTemp;
vaTemp.ChangeType(VT_BSTR, &vaTimeDate);
CString str = vaTemp.bstrVal;
TRACE("date = %s\n", str);
```

И последнее замечание о параметрах *Invoke*: у функции, вызываемой через *IDispatch*, могут быть необязательные (optional) параметры. Если компонент объявляет тип последних параметров (крайних справа) как *VARIANT*, то клиент не обязан передавать их. При таком вызове (без определения значения необязательного параметра) поле *vt* объекта *VARIANT* на стороне компонента будет содержать VT_ERROR.

Примеры Automation

В оставшейся части главы мы познакомимся с пятью примерами. Первые три — компоненты Automation: EXE-компонент без пользовательского интерфейса, DLL-компонент и EXE-компонент как SDI-приложение, допускающее одновременный запуск нескольких копий. К каждой из этих программ приложена рабочая книга Microsoft Excel. Четвертый пример — это MFC-клиент Automation, управляющий тремя вышеупомянутыми компонентами, а также приложением Excel при помощи класса *COleDispatchDriver*. Последний пример — клиентская программа, в которой вместо класса *COleDispatchDriver* используется директива *#import*.

Пример Ex23a: EXE-компонент без пользовательского интерфейса

Ex23a — типичный пример использования Automation. Программа похожа на Autoclick — пример MDI-приложения с объектом-документом в качестве компонента Automation. (Пример в Autoclick легко найти в MFC Library Reference, выполнив поиск по слову AutoClik.) Пример Ex23a отличается от Autoclick отсутствием пользовательского интерфейса. В Ex23a один класс, поддерживающий Automation, и в первой версии программы один процесс поддерживает создание нескольких объектов Automation. Во второй версии при создании клиентом Automation каждого нового объекта запускается новый процесс.

В Ex23a компонент, написанный на C++, реализует обработку финансовых транзакций. VBA-программистам доступно создание приложений с мощным пользовательским интерфейсом, которые полагаются на правила аудита, заложенные в логику компонента Automation. В коммерческом варианте такого компонента мы скорее всего использовали бы базу данных, но Ex23a проще. Здесь реализованы банковские счета с двумя методами *Deposit* (положить на счет) и *Withdrawal* (снять со счета), а также доступные только для чтения свойством *Balance* (остаток на счете). Ясно, что метод *Withdrawal* не должен создавать отрицательного остатка. Для управления компонентом можно задействовать Excel (рис. 23-3).

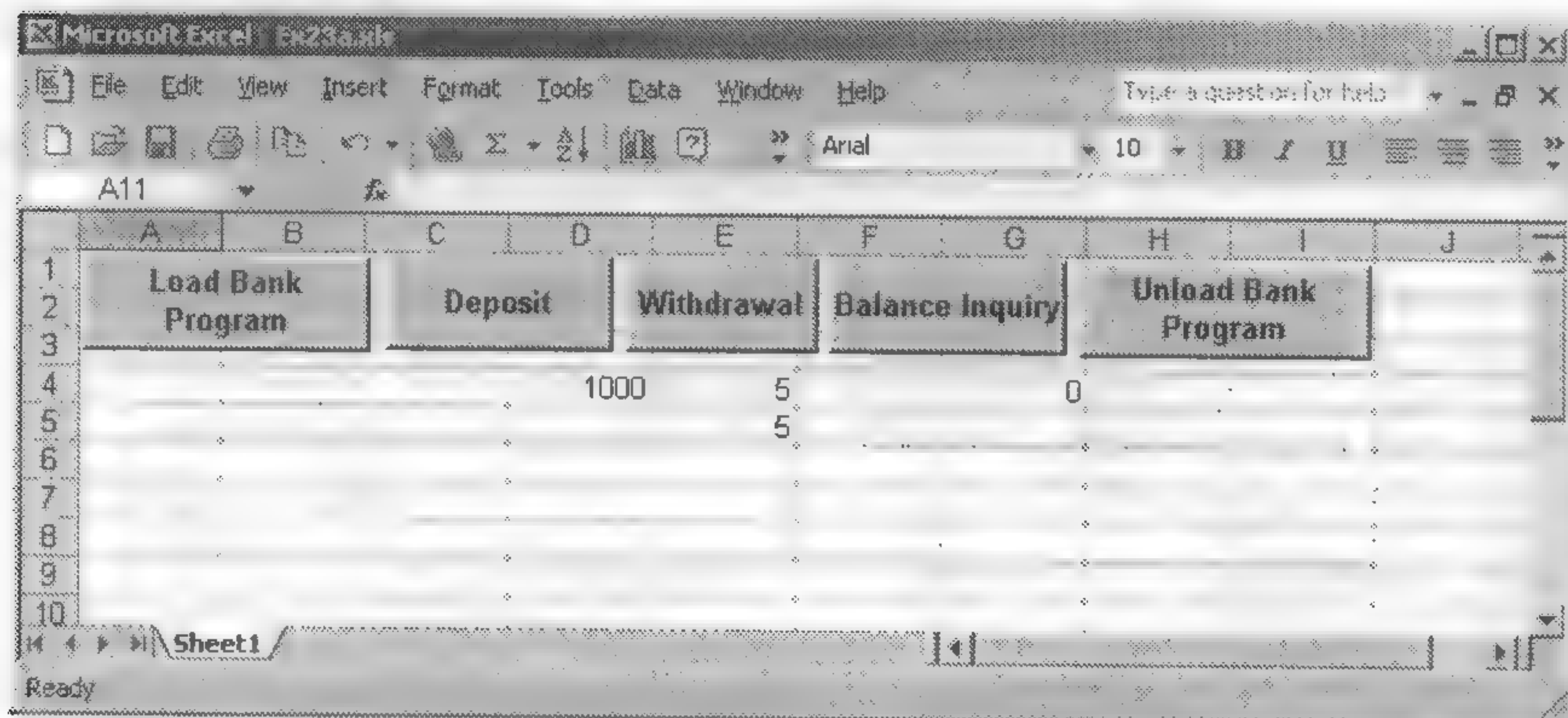


Рис. 23-3. *Рабочая книга Excel, управляющая компонентом Ex23a*

Итак, создадим программу Ex23a.

1. **Средствами MFC Application Wizard создайте проект Ex23a.** На странице Application Type установите переключатель в положение Dialog based. Сбросьте *все* флажки на страницах User Interface Features и Advanced Features, кроме флажка Automation на странице Advanced Features. В результате вы получите самое простое приложение, какое только умеет генерировать MFC Application Wizard.
2. **Уберите из проекта класс диалогового окна.** В Windows Explorer (Проводник) или в командной строке удалите файлы Ex23aDlg.cpp, Ex23aDlg.h, DlgProxy.cpp и DlgProxy.h. Уберите эти файлы и из проекта, удалив их в окне Solution Explorer. Отредактируйте файл Ex23a.cpp: удалите оператор *#include*, относящийся к классу диалогового окна, а также весь связанный с диалоговым окном код в *InitInstance*. В Resource View ликвидируйте диалоговый ресурс *IDD_EX23A_DIALOG*.
3. **Добавьте код поддержки Automation.** Благодаря установке флажка Automation в StdAfx.h появилась строка:

```
#include <afxdisp.h>
```

Функция *InitInstance* содержит теперь код инициализации COM. Добавьте в нее (в файле Ex23a.cpp) оператор return TRUE:

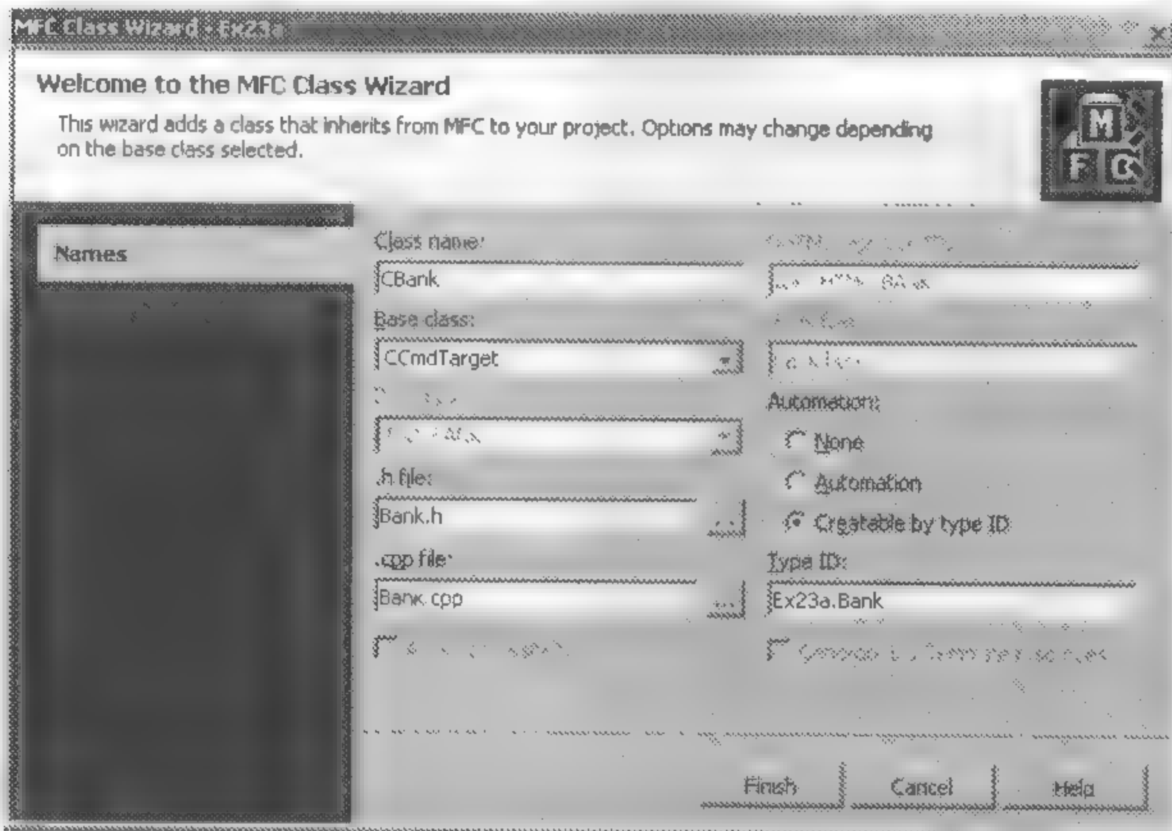
```
BOOL CEx23aApp::InitInstance()
{
```

```
    CWinApp::InitInstance();
    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
    // Parse command line for automation or reg/unreg switches.
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // App was launched with /Embedding or /Automation switch.
    // Run app as automation server.
    if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
    {
        // Register class factories via CoRegisterClassObject().
        COleTemplateServer::RegisterAll();
        return TRUE;

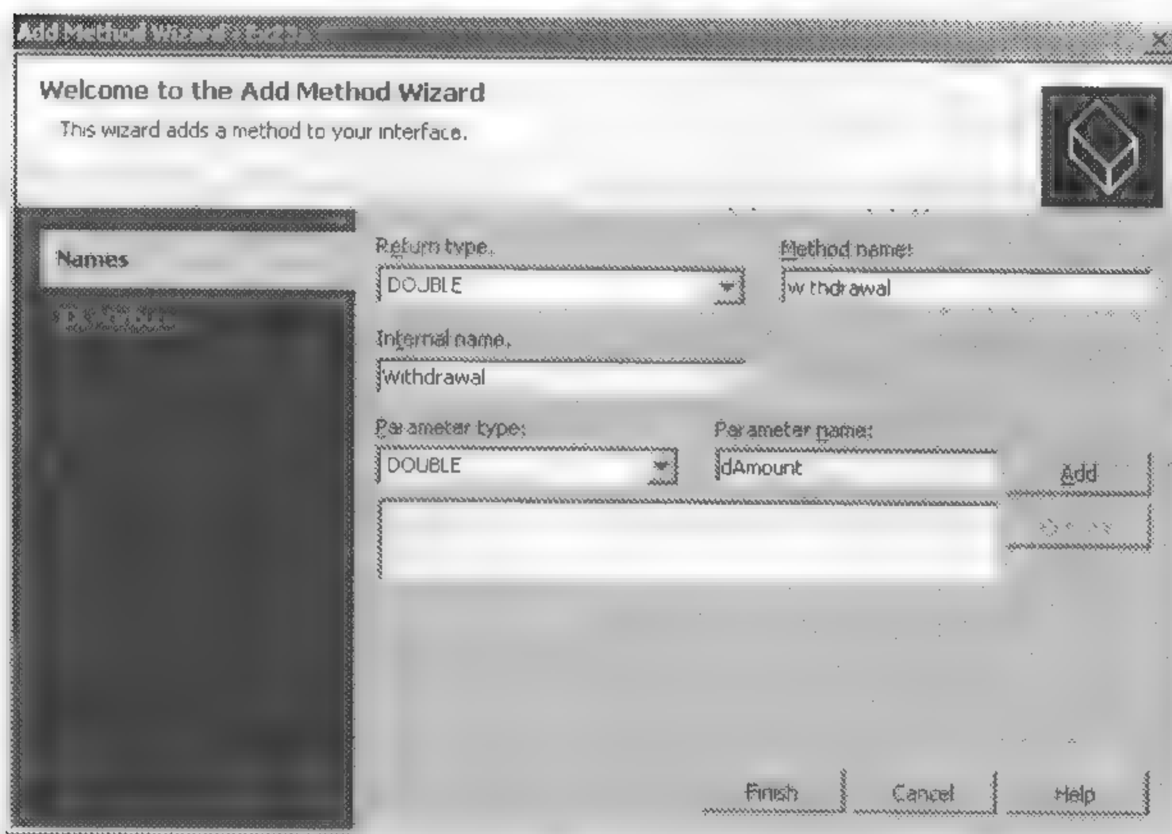
}
    // App was launched with /Unregserver or /Unregister switch.
    // Remove entries from the registry.
    else if (cmdInfo.m_nShellCommand ==
    CCommandLineInfo::AppUnregister)
    {
        COleObjectFactory::UpdateRegistryAll(FALSE);
        AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor);
        return FALSE;
    }
    // App was launched standalone or with other switches
    // (e.g. /Register or /Regserver). Update registry entries,
    // including typelibrary.
    else
    {
        COleObjectFactory::UpdateRegistryAll();
        AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid);
        if (cmdInfo.m_nShellCommand ==
            CCommandLineInfo::AppRegister)
            return FALSE;
    }
    return FALSE;
}
```

4. **Средствами мастера Add Class Wizard добавьте класс *CBank*:**



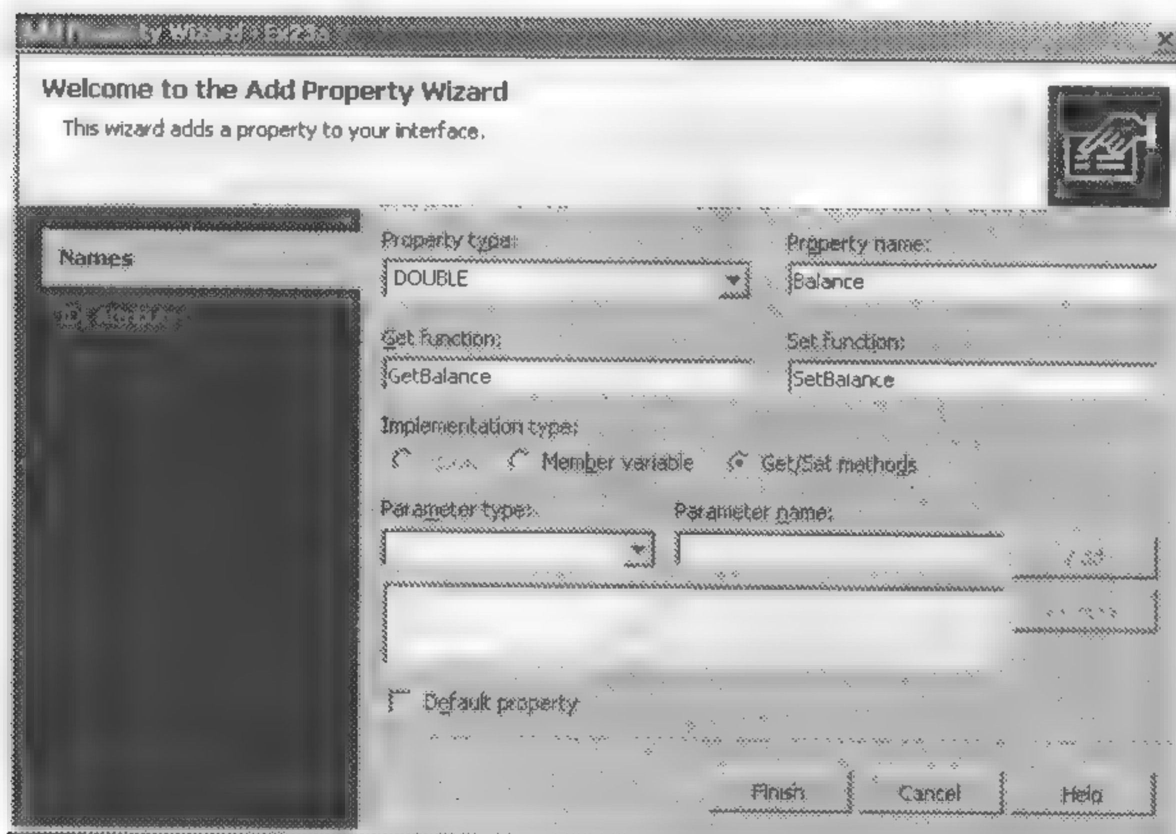
Обязательно установите переключатель в положение Createable by type ID.

5. **С помощью мастера Add Class Wizard добавьте два метода и свойство.** Откройте окно Class View, разверните узел библиотеки Ex23a и щелкните правой кнопкой узел *IBank*. Вы увидите команды Add Method и Add Property; добавьте метод *Withdrawal*:



Параметр *dAmount* задает сумму, снимаемую со счета, а в возвращаемом значении сообщается фактически снятая сумма. Если вы попробуете снять со счета $100, на котором только $60, функция вернет значение 60.

Добавьте аналогичный метод *Deposit*, возвращающий значение типа *void*, а потом — свойство *Balance*:



Мы могли бы напрямую обращаться к переменной-члену компонента, но тогда нам не удалось бы обеспечить доступ к нему в режиме только для чтения. Поэтому мы выбираем переключатель Get/Set methods и превращаем функцию *SetBalance* в ничего не делающую заглушку.

6. **Добавьте в класс *CBank* открытую переменную-член *m_dBalance* типа *double*.** Поскольку мы выбрали для свойства Balance параметр Get/Set methods, Add Property Wizard не сгенерирует переменную-член. Переменную следует объявить в файле Bank.h и инициализировать ее в конструкторе, расположенном в файле Bank.cpp, присвоив ей значение *0.0*.
7. **Отредактируйте сгенерированные функции для методов и свойств.** Добавьте выделенный код:

```
DOUBLE CBank::Withdrawal(DOUBLE dAmount)
{
    AFX_MANAGE_STATE(AfxGetAppModuleState());
    if (dAmount < 0.0) {
        return 0.0;
    }
    if (dAmount <= m_dBalance) {
        m_dBalance -= dAmount;
        return dAmount;
    }
    double dTemp = m_dBalance;
    m_dBalance = 0.0;
    return dTemp;
}
void CBank::Deposit(DOUBLE dAmount)
{
    AFX_MANAGE_STATE(AfxGetAppModuleState());
    if (dAmount < 0.0) {
        return;
    }
    m_dBalance += dAmount;
}
DOUBLE CBank::GetBalance(void)
```

```
{
    AFX_MANAGE_STATE(AfxGetAppModuleState());
    return m_dBalance;
}
void CBank::SetBalance(DOUBLE newVal)
{
    AFX_MANAGE_STATE(AfxGetAppModuleState());
    TRACE("Sorry, Dave, I can't do that!\n");
}
```

8. Соберите программу Ex23a и запустите ее один раз на исполнение, чтобы зарегистрировать компонент.
9. **Подготовьте пять макросов Excel в новом файле Ex23a.xls.** Вот их текст:

```
Dim Bank As Object
Sub LoadBank()
    Set Bank = CreateObject("Ex23a.Bank")
End Sub

Sub UnloadBank()
    Set Bank = Nothing
End Sub

Sub DoDeposit()
    Range("D4").Select
    Bank.Deposit (ActiveCell.Value)
End Sub

Sub DoWithdrawal()
    Range("E4").Select
    Dim Amt
    Amt = Bank.Withdrawal(ActiveCell.Value)
    Range("E5").Select
    ActiveCell.Value = Amt
End Sub

Sub DoInquiry()
    Dim Amt
    Amt = Bank.Balance()
    Range("G4").Select
    ActiveCell.Value = Amt
End Sub
```

10. **Подготовьте рабочую тетрадь Excel (см. рис. 23-3).** Закрепите макросы за кнопками (воспользуйтесь правой кнопкой и контекстным меню).
11. **Протестируйте банковский компонент Ex23a.** Щелкните кнопку Load Bank Program, введите вносимую на счет сумму в ячейку D4 и щелкните кнопку Deposit. Щелкните кнопку Balance Inquiry — в ячейке G4 должно появиться значение остатка. Введите в ячейку E5 снимаемую со счета сумму и щелкните кнопку Withdrawal. Чтобы узнать текущий остаток на счете, опять щелкните кнопку Balance Inquiry.

Примечание Иногда придется щелкать кнопки два раза подряд. Первый щелчок возвращает фокус на таблицу, и только второй запускает макрос. Курсор в виде песочных часов подскажет, что макрос исполняется.

Что происходит в этой программе? Рассмотрим функцию *CEx23aApp::InitInstance*. Если ее запустить прямо из Windows, она обновляет реестр, сообщает об этом в информационном окне и завершается. Функция *COleObjectFactory::UpdateRegistryAll* выполняет поиск глобальных объектов — фабрик классов; макрос *IMPLEMENT_OLECREATE* из класса *CBank* как раз и определяет такой объект. (Строка, содержащая *IMPLEMENT_OLECREATE_FLAGS*, сгенерирована потому, что мы выбрали для *CBank* параметр Createable by type ID.) В реестр добавляются уникальный идентификатор класса и программный идентификатор *Ex23aBANK*.

Когда Excel вызывает *CreateObject*, COM загружает программу Ex23a, содержащую глобальную фабрику для объектов *CBank*; затем COM вызывает функцию *CreateInstance* объекта-фабрики для создания объекта *CBank* и возвращает указатель на *IDispatch*. Если пропустить несущественные подробности (и показанные ранее функции для методов и свойств), то сгенерированный мастером Add Class Wizard класс *CBank* выглядит так:

```
#pragma once
// CBank command target
class CBank : public CCmdTarget
{
   DECLARE_DYNCREATE(CBank)
public:
   CBank();
   virtual ~CBank();
   virtual void OnFinalRelease();
   DOUBLE m_dBalance;
protected:
   DECLARE_MESSAGE_MAP()
   DECLARE_OLECREATE(CBank)
   DECLARE_DISPATCH_MAP()
   DECLARE_INTERFACE_MAP()
   DOUBLE Witndrawal(DOUBLE dAmount);
   enum
   {
      dispidBalance = 3, dispidDeposit = 2L, dispidWithdrawal = 1L
   };
   void Deposit(DOUBLE dAmount);
   DOUBLE GetBalance(void);
   void SetBalance(DOUBLE newVal);
};
```

А вот код, автоматически сгенерированный Add Class Wizard в файле bank.cpp:

```
// Bank.cpp : implementation file
//
#include "stdafx.h"
#include "Ex23a.h"
#include "Bank.h"
```

```
// CBank
IMPLEMENT_DYNCREATE(CBank, CCmdTarget)
CBank::CBank()
{
   EnableAutomation();
   // To keep the application running as long as an OLE automation
   // object is active, the constructor calls AfxOleLockApp.
   AfxOleLockApp();
   m_dBalance = 0.0;
}
CBank::~CBank()
{
   // To terminate the application when all objects created with
   //  with OLE automation, the destructor calls AfxOleUnlockApp.
   AfxOleUnlockApp();
}
void CBank::OnFinalRelease()
{
   // When the last reference for an automation object is released
   // OnFinalRelease is called.  The base class will automatically
   // delete the object.  Add additional cleanup required for your
   // object before calling the base class.
   CCmdTarget::OnFinalRelease();
}
BEGIN_MESSAGE_MAP(CBank, CCmdTarget)
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CBank, CCmdTarget)
 DISP_FUNCTION_ID(CBank, "Withdrawal", dispidWithdrawal,
                  Withdrawal, VT_R8, VTS_R8)
 DISP_FUNCTION_ID(CBank, "Deposit", dispidDeposit,
                  Deposit, VT_EMPTY, VTS_R8)
 DISP_PROPERTY_EX_ID(CBank, "Balance", dispidBalance,
                   GetBalance, SetBalance, VT_R8)
END_DISPATCH_MAP()

// Note: we add support for IID_IBank to support typesafe binding
// from VBA.  This IID must match the GUID that is attached to the
// dispinterface in the .IDL file.

// {8BAD2B0C-62CC-4952-811C-C736DA06858E}
static const IID IID_IBank =
  { 0x8BAD2B0C, 0x62CC, 0x4952,
   { 0x81, 0x1C, 0xC7, 0x36, 0xDA, 0x6, 0x85, 0x8E } };

BEGIN_INTERFACE_MAP(CBank, CCmdTarget)
   INTERFACE_PART(CBank, IID_IBank, Dispatch)
END_INTERFACE_MAP()

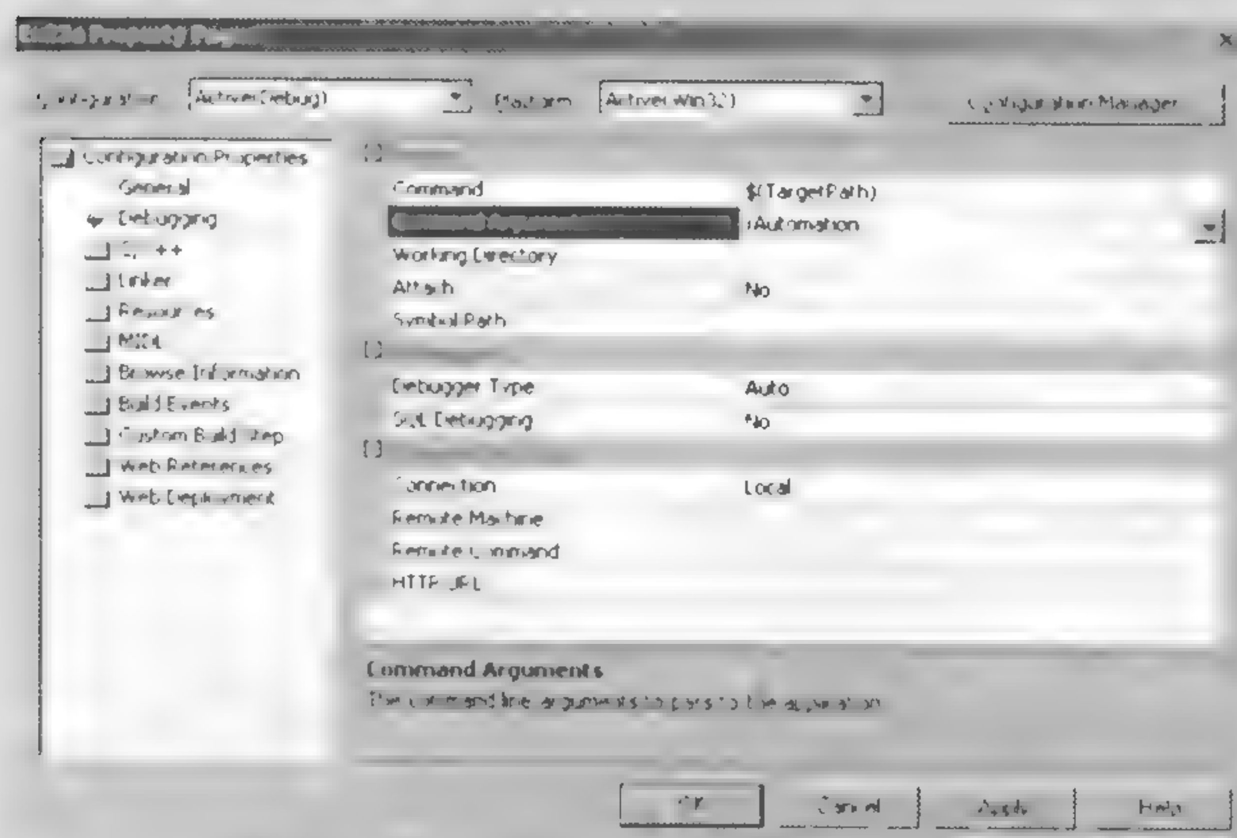
// {3EC6FA59-9F9F-4619-9F62-BA5FE37176F0}
IMPLEMENT_OLECREATE_FLAGS(CBank, "Ex23a.Bank",
```

```
    afxRegApartmentThreading, 0x3ec6fa59, 0x9f9f,
    0x4619, 0x9f, 0x62, 0xba, 0x5f, 0xe3, 0x71,
    0x76, 0xf0)
// CBank message handlers
:
```

Эта первая версия программы Ex23a работает в режиме единственного процесса, как и программа Autoclick. Если второй клиент Automation запрашивает второй объект *CBank*, COM вновь вызывает функцию *CreateInstance* фабрики класса, и существующий процесс конструирует в куче новый объект *CBank*. Это можно проверить, создав копию рабочей книги Ex23a.xls под другим именем и загрузив и копию, и оригинал. Щелкните кнопку Load Bank Program в обеих книгах и проследите за сообщениями в окне Debug. Процедура *InitInstance* должна вызываться только раз.

Отладка программы EXE-компонента

При запуске EXE-компонента клиент Automation указывает в командной строке параметр */Embedding*. Для отладки компонента вы должны сделать то же самое. Щелкните правой кнопкой проект в окне Solution Explorer и в контекстном меню выберите Properties. В открывшемся окне выберите папку Debugging и в поле Command Arguments введите **/Automation** (или **/Embedding**).



После выбора команды Start в меню Debug или нажатия клавиши F5 программа запустится и станет ждать, когда ее активизирует клиент. Теперь можно запустить из Windows и клиентскую программу (если она еще не запущена) и создать с ее помощью объект компонента. В результате запущенная в отладчике программа должна сконструировать объект. Неплохая идея — включить оператор *TRACE* в конструктор объекта.

Не забудьте зарегистрировать программу компонента, иначе клиент ее не найдет. Это значит, что вы должны один раз запустить ее *без параметра /Automation* (или */Embedding*). Многие клиенты не отслеживают изменения в реестре, поэтому, если ваш клиент уже работал в момент регистрации компонента, его, может быть, придется перезапустить.

Пример Ex23b: DLL-компонент Automation

Ex23a можно легко переделать из EXE-варианта в DLL. Класс *CBank* при этом не изменится, а макросы Excel будут очень похожи. Однако интереснее написать новое приложение, на этот раз с минимальным пользовательским интерфейсом. Применим в нем модальное диалоговое окно, поскольку на практике это максимум того, что можно сравнительно беспроблемно добавить в DLL Automation.

Программа Ex23b довольно проста. Класс компонента Automation, определяемый по регистрируемому имени Ex23b.Auto, имеет такие свойства и методы:

Название	Описание
LongData	Свойство типа длинное целое
TextData	Свойство типа *VARIANT*
DisplayDialog	Метод без параметров, возвращает *BOOL*

DisplayDialog отображает диалоговое окно для ввода данных (рис. 23-4). Макрос Excel передает DLL значения двух ячеек, а потом обновляет эти же ячейки новыми значениями.

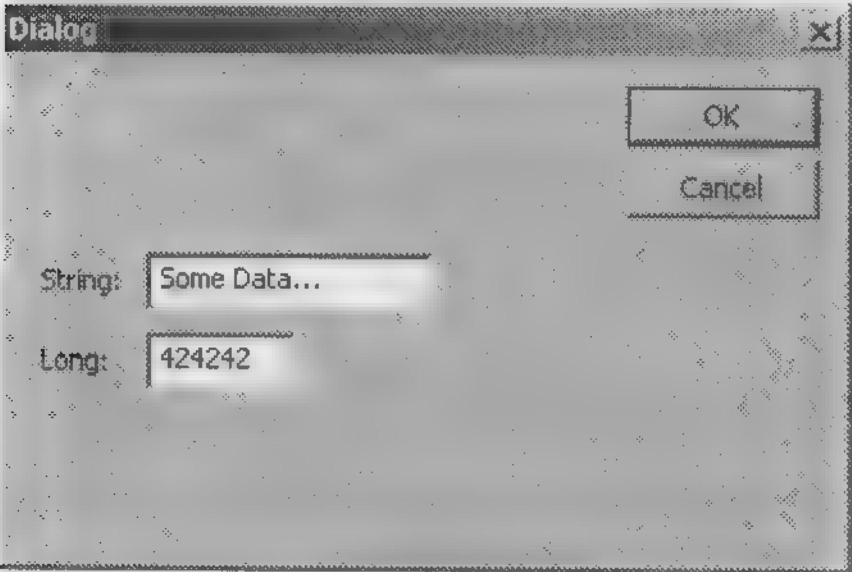


Рис. 23-4. *Диалоговое окно DLL-библиотеки Ex23b в действии*

Передача параметров по ссылке

До сих пор вы видели передачу параметров в VBA *по значению*. Надо сказать, что в VBA весьма странные правила вызова методов: один параметр у метода — можно ставить скобки, более одного — нельзя (но если вы используете возвращаемое значение функции, скобки *необходимы* в любом случае). Вот несколько примеров передачи в VBA строкового параметра по значению:

```
Object.Method1 par1, "text"
Object.Method2("text")
Dim s As String
s = "text"
Object.Method2(s)
```

И все же иногда VBA передает параметр как адрес (по ссылке). В следующем примере строка передается по ссылке:

см. след. стр.

```
Dim s as String
s = "text"
Object.Method1 parm1 := s
```

Правила, применяемые VBA по умолчанию, можно переопределить, указывая перед параметром ключевые слова *ByVal* (по значению) или *ByRef* (по ссылке). Компонент заранее не знает, как будет передан параметр: по значению или по ссылке, — так что он должен быть готов к обоим вариантам. Для этого ему придется проверять, не установлен ли признак *VT_BYREF* в поле *vt*. Вот пример реализации метода, допускающего передачу строки (в формате *VARIANT*) как по ссылке, так и по значению:

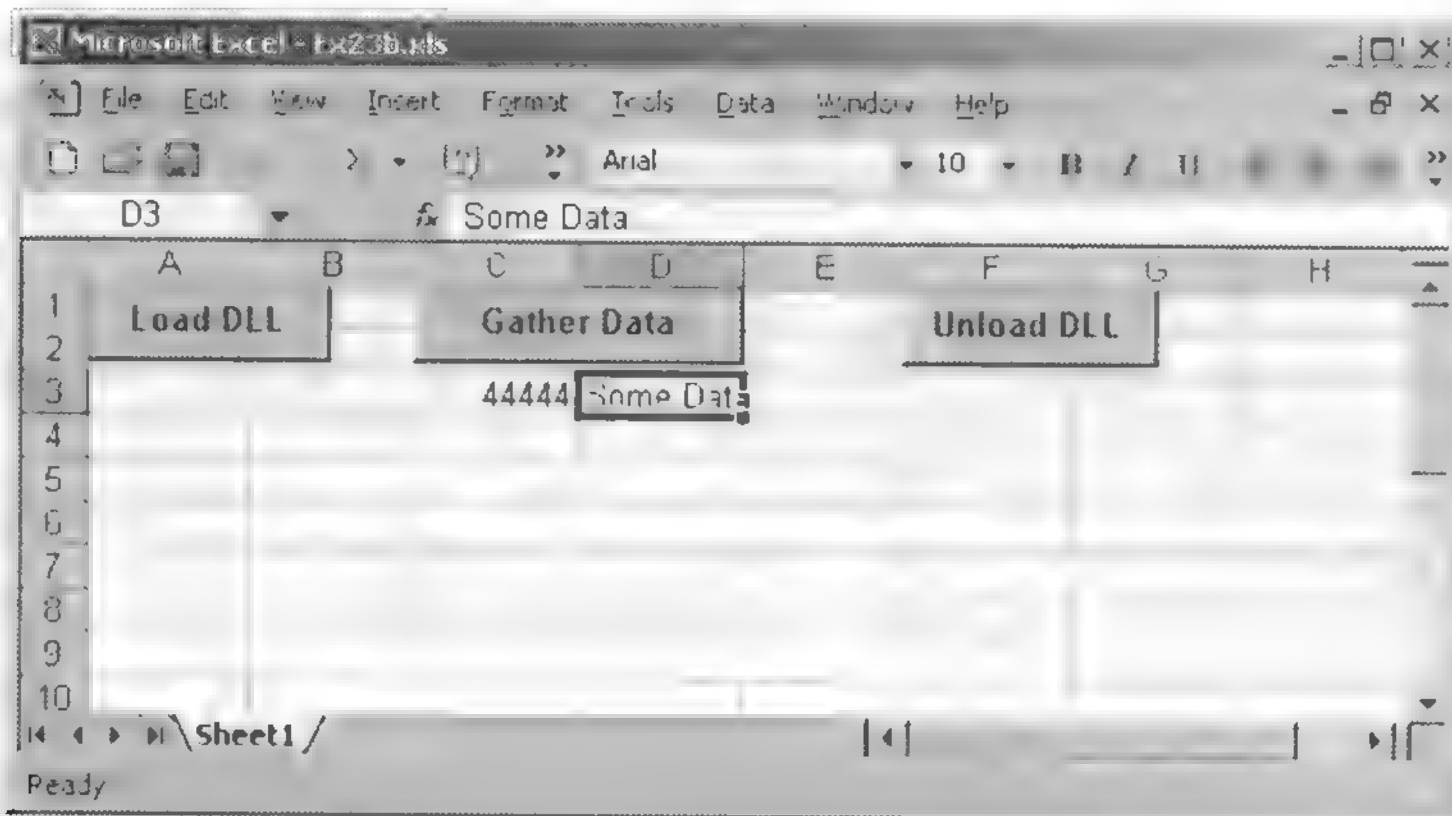
```
void CMyServer::Method(long nParm1, const VARIANT& vaParm2)
{
   CString str;
   if ((vaParm2.vt & 0x7f) == VT_BSTR) {
      if ((vaParm2.vt & VT_BYREF) != 0)
         str = *(vaParm2.pbstrVal); // по ссылке
      else
         str = vaParm2.bstrVal; // по значению
   }
   AfxMessageBox(str);
}
```

Если объявить параметр как BSTR, MFC выполнит все преобразования сама. Пусть ваша клиентская программа передает ссылку на BSTR во внешний компонент, который изменяет значение строки. Поскольку компонент не имеет доступа к памяти клиентского процесса, COM должна скопировать строку для компонента, а затем сделать еще одну копию для клиента по завершении работы функции. Поэтому, объявляя передачу параметров по ссылке, помните, что передача ссылок через *IDispatch* сильно отличается от передачи ссылок в C++.

Этот пример изначально сгенерирован средствами MFC DLL Wizard как обычная MFC DLL с установленными переключателем Regular DLL using shared MFC DLL и флажком Automation. Вот как подготовить и протестировать DLL-сервер Ex23b, используя код с компакт-диска.

1. **Откройте в Visual Studio .NET решение \vcppnet\Ex23b\Ex23b.sln.** Соберите проект.
2. **Зарегистрируйте DLL.** Можете использовать для этой цели программу RegComp из каталога \vcpp32\Regcomp\Release компакт-диска, которая открывает диалоговое окно, упрощающее выбор DLL-файла. Впрочем, вы вправе применить стандартную утилиту Regsvr32.exe.
3. **Откройте Excel и загрузите файл \vcppnet\Ex23b\Ex23b.xls.** Введите целое число в ячейку C3 и какой-нибудь текст в ячейку D3 (см. рис. на следующей странице).

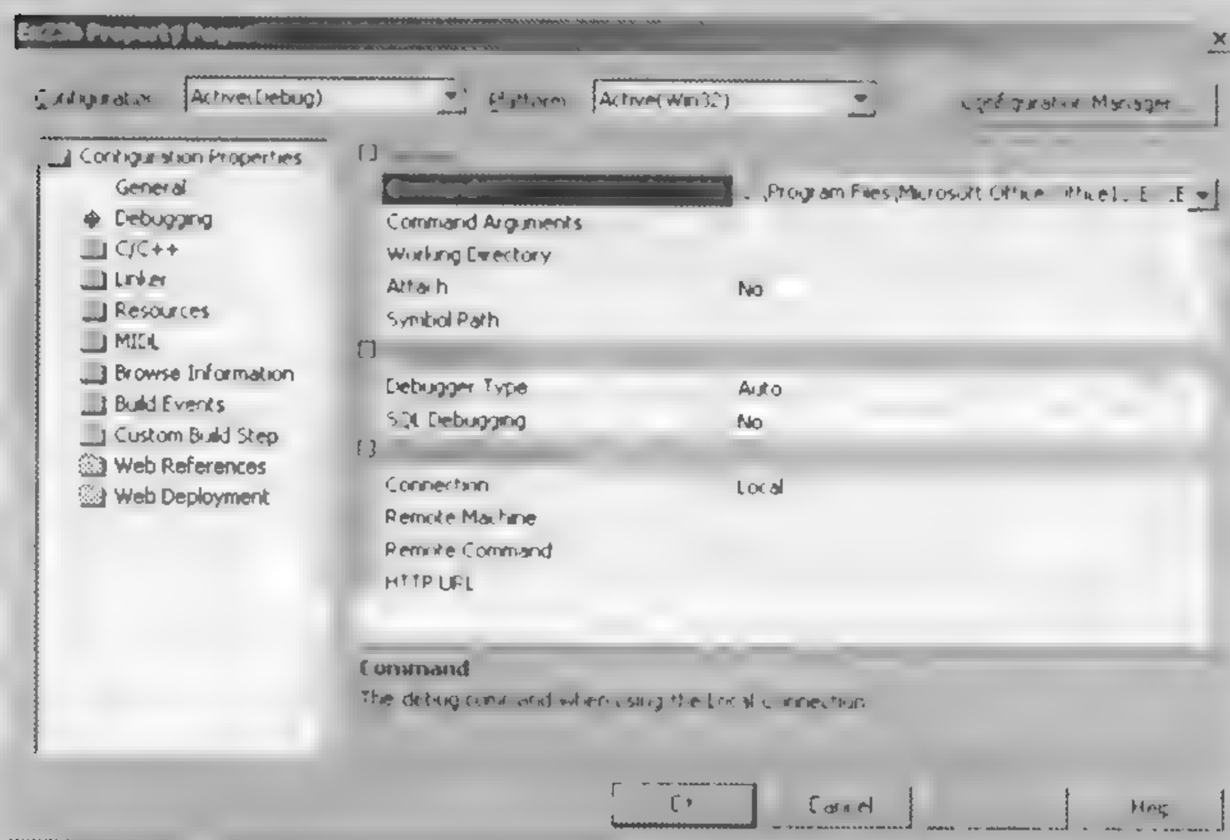
   Щелкните кнопку Load DLL, затем — кнопку Gather Data. Введите данные, щелкните кнопку OK и посмотрите на новые значения, появившиеся в электронной таблице.



4. **Щелкните кнопку Unload DLL.** Если вы запустили DLL (и Excel) из отладчика, просмотрите окно Debug и убедитесь, что в DLL вызвана функция *ExitInstance*.

Отладка DLL-компонента

Для отладки DLL надо сообщить отладчику, какой EXE-файл он должен загрузить. Щелните правой кнопкой проект в окне Solution Explorer и в контекстном меню выберите Properties. В открывшемся окне Property Pages выберите папку Debugging и в поле Command введите полное имя контроллера (в том числе расширение .exe):



Нажатие клавиши F5 запустит контроллер (что приведет и к загрузке вашей DLL), и тот будет ждать, пока вы не активизируете компонент.

Неплохо в конструктор объекта компонента включить оператор *TRACE*. Но не забудьте: прежде чем клиент сможет загрузить DLL-компонент, его надо зарегистрировать.

Есть и другой вариант. Если вам доступен исходный код клиентской программы, можно запустить в отладчике ее. Когда клиент загрузит DLL-компонент, вы увидите соответствующие сообщения операторов *TRACE*.

Теперь обратимся к коду Ex23b. Как и в MFC EXE, в обычной DLL на базе MFC есть класс приложения (производный от *CWinApp*), а также глобальный объект-приложение. Переопределенная функция *InitInstance* в Ex23b.cpp выглядит так:

```
BOOL CEx23bApp::InitInstance()
{
    TRACE("CEx23bApp::InitInstance\n");
    CWinApp::InitInstance();

    // Register all OLE server (factories) as running.  This enables the
    //  OLE libraries to create objects from other applications.
    COleObjectFactory::RegisterAll();

    return TRUE;
}
```

В программе есть код для трех стандартных экспортируемых функций COM DLL:

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllGetClassObject(rclsid, riid, ppv);
}
// DllCanUnloadNow - Allows COM to unload DLL
STDAPI DllCanUnloadNow(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllCanUnloadNow();
}
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll())
        return SELFREG_E_CLASS;

    return S_OK;
}
// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    if (!AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor))
        return SELFREG_E_TYPELIB;

    if (!COleObjectFactory::UpdateRegistryAll(FALSE))
```

```
        return SELFREG_E_CLASS;

    return S_OK;
}
```

Код класса *CPromptDlg* находится в файле PromptDlg.cpp. Но это стандартный класс производный от *CDialog*. Файл PromptDlg.h содержит заголовок этого класса. Нам более интересен класс компонента Automation *CEx23bAuto*, изначально сгенерированный Add Class Wizard (с параметром Createable by type ID). Этот класс доступен COM под программным идентификатором Ex23b.Ex23bAuto. Так выглядит листинг заголовочного файла Ex23bAuto.h.

Ex23bAuto.h

```
[illegible]
```

А это файл реализации Ex23bAuto.cpp.

Ex23bAuto.cpp

```
[illegible]
```

см. след. стр.

```
#include "[illegible].h"

// CEx25bAuto
IMPLEMENT_DYNCREATE(CEx25bAuto, CCmdTarget)
CEx25bAuto::CEx25bAuto()
{
   EnableAutomation();
   // To keep the application running as long as an OLE automation
   // object is active, the constructor calls AfxOleLockApp.

   ::VariantInit(&m_vaTextData); // necessary initialization
   m_lData = 0;

   AfxOleLockApp();
}
CEx25bAuto::~CEx25bAuto()
{
   // To terminate the application when all objects created with
   // with OLE automation, the destructor calls AfxOleUnlockApp.
   AfxOleUnlockApp();
}
void CEx25bAuto::OnFinalRelease()
{
   // When the last reference for an automation object is released
   // OnFinalRelease is called. The base class will automatically
   // deletes the object. Add additional cleanup required for your
   // object before calling the base class.
   CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CEx25bAuto, CCmdTarget)
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CEx25bAuto, CCmdTarget)
   DISP_PROPERTY_NOTIFY(CEx25bAuto, "LongData", m_lData,
                        OnLongDataChanged, VT_I4)
   DISP_PROPERTY_NOTIFY(CEx25bAuto, "TextData", m_vaTextData,
                        OnTextDataChanged, VT_VARIANT)
   DISP_FUNCTION(CEx25bAuto, "DisplayDialog", DisplayDialog,
                 VT_BOOL, VTS_NONE)
END_DISPATCH_MAP()

// Note: we add support for IID_IEx25bAuto to support typesafe binding
// from VBA. This IID must match the GUID that is attached to the
// dispinterface in the .ODL file.

// {[illegible]}
static const IID IID_IEx25bAuto =
   { [illegible], [illegible], [illegible], { [illegible], [illegible], [illegible], [illegible], [illegible],
     [illegible], [illegible], [illegible] } };
```

[illegible]

см. след. стр.

```
        bRet = FALSE;
        pe->Delete();
    }
    AfxUnlockTempMaps();
    return bRet;
}
```

Свойства *LongData* и *TextData* представлены переменными-членами *m_lData* и *m_vaTextData*, инициализируемыми в конструкторе. Когда вы добавили в мастере Add Property Wizard свойство LongData, была определена уведомляющая функция *OnLongDataChanged*. Она вызывается всякий раз, когда контроллер изменяет значение свойства. Такие функции можно задать только для свойств, представленных переменными-членами. Не путайте эти уведомления с теми, что ActiveX-элементы посылают своему контейнеру при изменении связанного свойства.

Функция-член *DisplayDialog*, реализующая одноименный метод, проста, если не считать того, что для очистки указателей на временные объекты нужны функции *AfxLockTempMaps* и *AfxUnlockTempMaps* (обычно такая операция осуществляется в холостом цикле EXE-программы).

А как насчет VBA-кода в Excel? Вот три макроса и глобальные объявления:

```
Dim Dllcomp As Object
Private Declare Sub CoFreeUnusedLibraries Lib "OLE32" ()

Sub LoadDllComp()
    Set Dllcomp = CreateObject("Ex23b.Ex23bAuto")
    Range("C3").Select
    Dllcomp.LongData = Selection.Value
    Range("D3").Select
    Dllcomp.TextData = Selection.Value
End Sub

Sub RefreshDllComp() 'Кнопка Gather Data
    Range("C3").Select
    Dllcomp.LongData = Selection.Value
    Range("D3").Select
    Dllcomp.TextData = Selection.Value
    Dllcomp.DisplayDialog
    Range("C3").Select
    Selection.Value = Dllcomp.LongData
    Range("D3").Select
    Selection.Value = Dllcomp.TextData
End Sub

Sub UnloadDllComp()
    Set Dllcomp = Nothing
    Call CoFreeUnusedLibraries
End Sub
```

Первая строка в *LoadDllComp* создает объект компонента в соответствии с зарегистрированным именем Ex23b.Ex23bAuto. Макрос *RefreshDllComp* обращается

к свойствам этого объекта *TextData* и *LongData*. При первом запуске *LoadDllComp* загружается DLL, и создается объект *Ex23bAuto*. При втором происходит нечто любопытное: создается второй объект, а первый удаляется. Запустив *LoadDllComp* из другой копии рабочей книги, вы получите два отдельных объекта *Ex23bAuto*. Конечно, в памяти только одна копия Ex23b.dll, если только вы не запустили несколько экземпляров Excel.

Присмотримся к макросу *UnloadDllComp*. Следующий оператор заставляет Excel отсоединить DLL, но из адресного пространства Excel она не выгружается, а значит, функция компонента *ExitInstance* не вызывается.

```
Set Dllcomp = Nothing
```

Функция *CoFreeUnusedLibraries* вызывает для каждой компонентной DLL экспортируемую функцию *DllCanUnloadNow* и, если функция возвращает *TRUE*, освобождает DLL. Программы на базе MFC вызывают *CoFreeUnusedLibraries* в холостом цикле (с минутной задержкой), но Excel этого не делает. Вот почему *UnloadDllComp* должна вызывать *CoFreeUnusedLibraries* после отсоединения компонента.

Пример Ex23c: SDI-приложение Automation в виде EXE-компонента с пользовательским интерфейсом

Этот пример компонента Automation иллюстрирует применение класса документа в качестве компонентного класса в SDI-приложении, где для каждого объекта запускается отдельный процесс. Здесь также демонстрируется индексируемое свойство и метод, конструирующий новый COM-объект.

В первом компоненте Automation — Ex23a — не было пользовательского интерфейса. Глобальная фабрика классов создавала объект *CBank*, который делал всю работу. А если нужен EXE-компонент с окном? Если вы уже разбираетесь в архитектуре «документ-вид», поддерживаемой MFC, то наверняка не откажетесь задействовать все ее преимущества.

Допустим, вы создали обычное MFC-приложение и добавили к нему класс (например, *CBank*), который можно создать посредством COM. Как связать объект *CBank* с документом и его представлением? Из функции-члена класса *CBank* вы могли бы перейти через объект-приложение и основное окно к текущему документу или объекту «вид», но в MDI-приложении это не так просто, особенно при наличии нескольких объектов и документов. Есть способ получше: предусмотреть в классе документа возможность его создания по COM — MFC Application Wizard поддерживает это и в MDI-, и в SDI-приложениях.

MDI-приложение Autoclick демонстрирует, как COM инициирует создание нового документа, окна представления и дочернего окна-рамки всякий раз, когда контроллер автоматизации создает новый объект компонента. Ну, а поскольку Ex23c — SDI-приложение, Windows при создании клиентом нового объекта запускает новый экземпляр программы. Сразу после запуска программы COM при помощи каркаса приложения MFC создает не только документ, поддерживающий Automation, но и окно представления и основное окно-рамку.

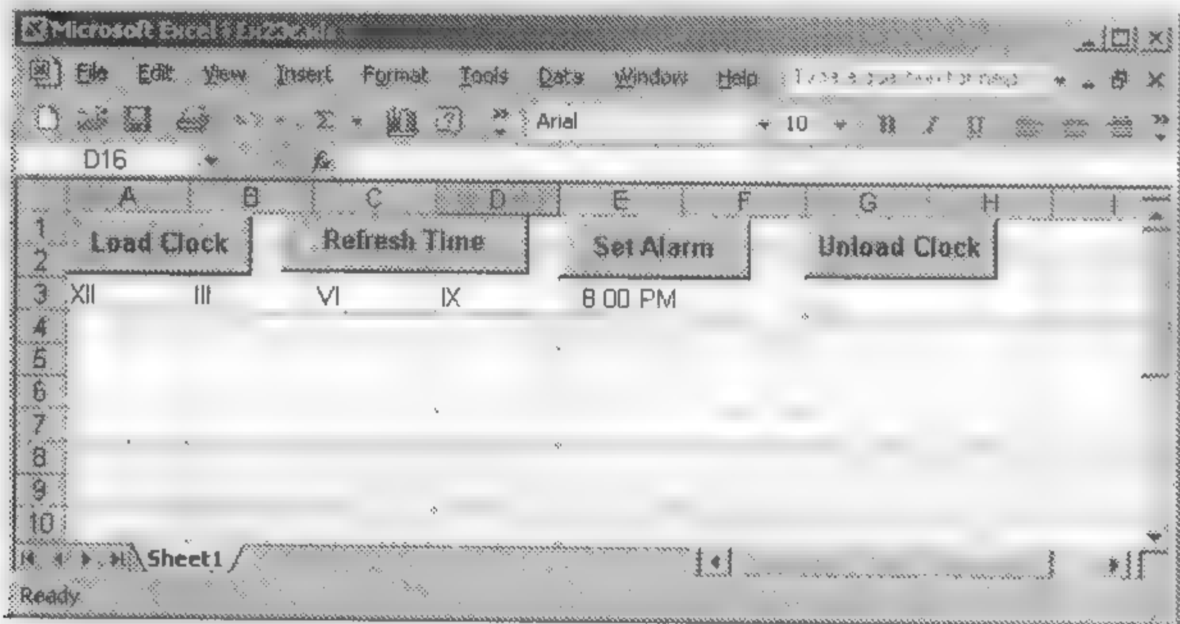
Поэкспериментируем с приложением Ex23c, изначально сгенерированным MFC DLL Wizard с установленным флажком Automation. Оно представляет собой про-

грамму-будильник с оконным пользовательским интерфейсом, предназначенную для управления клиентом Automation, таким как Excel. Вот свойства и методы Ex23c.

Название	Описание
Time	Свойство типа *DATE*, содержащее дату в формате COM (*m_vaTime*)
Figure	Индексируемое свойство типа *VARIANT* для четырех цифр на циферблате (*m_strFigure[]*)
RefreshWin	Метод, объявляющий окно представления недействительным и помещающий основное окно-рамку поверх других окон (*Refresh*)
ShowWin	Метод, отображающий основное окно приложения (*ShowWin*)
CreateAlarm	Метод, создающий объект *CAlarm* и возвращающий указатель его интерфейса *IDispatch* (*CreateAlarm*)

Чтобы создать и запустить программу Ex23c с компакт-диска, сделайте так.

1. **Откройте в Visual Studio .NET проект \vcppnet\Ex23c\Ex23c.sln.** Соберите проект, чтобы получить файл Ex23c.exe в подкаталоге Debug проекта.
2. **Запустите программу один раз для ее регистрации.** Программа может работать как автономное приложение или как компонент Automation. При запуске из Windows или из Visual Studio .NET она обновляет реестр и отображает циферблат часов с римскими цифрами XII, III, VI и IX в соответствующих позициях. Закройте приложение.
3. **Загрузите в Excel файл \vcppnet\Ex23c\Ex23c.xls.** Таблица выглядит так:

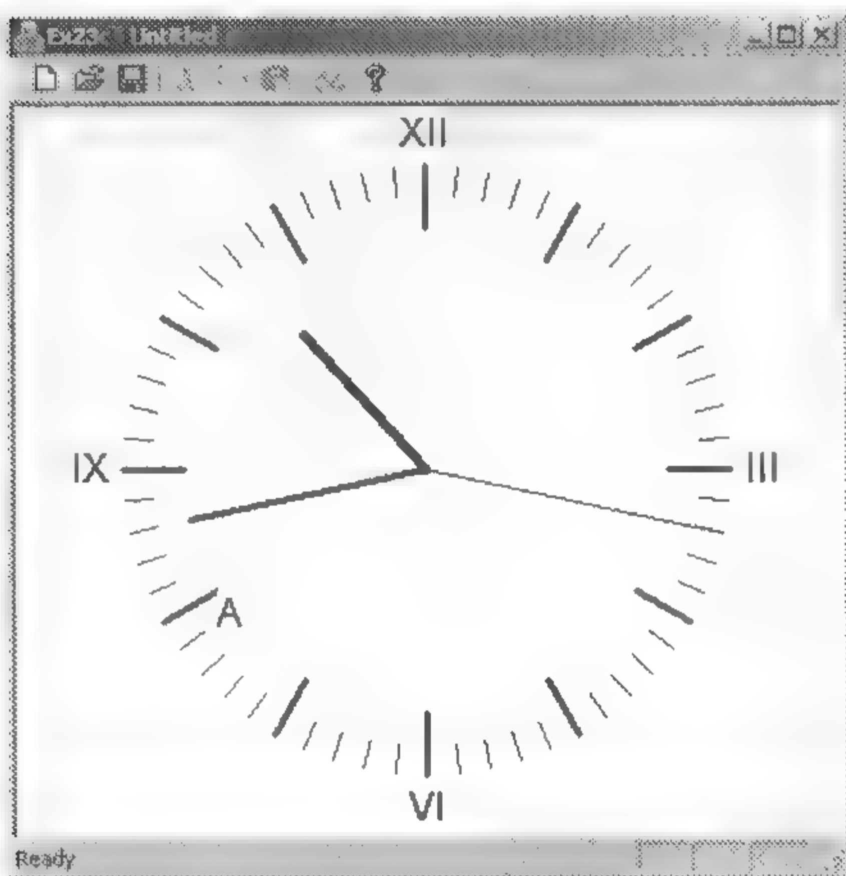


Щелкните кнопку Load Clock, затем дважды (два одинарных щелчка) — кнопку Set Alarm (после щелчка кнопки Load Clock может быть длительная задержка). На экране должны появиться часы с буквой *A* — индикатором того, что будильник установлен.

Если вы запустите компонент под управлением отладчика, сможете увидеть в окне Debug, когда была вызвана *InitInstance* и когда был создан объект-документ.

Если вас удивляет отсутствие меню в окне будильника, причина тому — наличие в *CMainFrame::PreCreateWindow* оператора:

```
cs.hMenu = NULL;
```



4. **Завершите программу Clock, а затем щелкните кнопку Unload Clock.** Если компонент запущен из отладчика, в окне Debug появится сообщение о вызове функции *ExitInstance*.

Наибольшую часть работы по созданию документа как компонента Automation проделал MFC Application Wizard. В производном классе приложения *CEx23cApp* он сгенерировал для компонента переменную-член:

```
public:
    COleTemplateServer m_server;
```

MFC-класс *COleTemplateServer* наследует *COleObjectFactory*. Он создает COM-объект «документ» при вызове клиентом *IClassFactory::CreateInstance*. Идентификатор класса хранится в глобальной переменной *clsid*, определенной в файле Ex23c.cpp. Программный идентификатор (Ex23c.Document) находится в строковом ресурсе *IDR_MAINFRAME*.

В функции *InitInstance* (файл Ex23c.cpp) MFC Application Wizard сгенерировал код, подключающий компонентный объект (документ) к шаблону документа:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CEx23cDoc),
    RUNTIME_CLASS(CMainFrame),    // main SDI frame window
    RUNTIME_CLASS(CEx23cView));
AddDocTemplate(pDocTemplate);
:
m_server.ConnectTemplate(clsid, pDocTemplate, TRUE);
```

Теперь все готово для того, чтобы COM и каркас приложений смогли создать документ вместе с окном представления и окном-рамкой. Однако формирование объектов не влечет автоматического вывода основного окна на экран. Это уже ваша работа — пишите соответствующий метод.

Следующий вызов *UpdateRegistry* в функции *InitInstance* обновляет реестр Windows на основе строкового ресурса *IDR_MAINFRAME*:

```
m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
```

Взгляните на карту диспетчеризации в файле Ex23cDoc.cpp с методам и свойствами класса *CEx23cDoc*. Обратите внимание: *Figure* — индексируемое свойство, которое Add Property Wizard может сгенерировать сам, если вы укажете ему подходящий параметр. Код, необходимый для функций *GetFigure* и *SetFigure*, мы рассмотрим позднее.

```
BEGIN_DISPATCH_MAP(CEx23cDoc, CDocument)
   DISP_PROPERTY_NOTIFY_ID(CEx23cDoc, "Time",
      dispidTime, m_time, OnTimeChanged, VT_DATE)
   DISP_FUNCTION_ID(CEx23cDoc, "ShowWin",
      dispidShowWin, ShowWin, VT_EMPTY, VTS_NONE)
   DISP_FUNCTION_ID(CEx23cDoc, "CreateAlarm",
      dispidCreateAlarm, CreateAlarm, VT_DISPATCH, VTS_DATE)
   DISP_FUNCTION_ID(CEx23cDoc, "RefreshWin",
      dispidRefreshWin, RefreshWin, VT_EMPTY, VTS_NONE)
   DISP_PROPERTY_PARAM_ID(CEx23cDoc, "Figure",
      dispidFigure, GetFigure, SetFigure, VT_VARIANT, VTS_I2)
END_DISPATCH_MAP()
```

Методы *RefreshWin* и *ShowWin* не представляют особого интереса, но метод *CreateAlarm* заслуживает особого внимания. Вот как выглядит соответствующая ему функция-член *CreateAlarm*:

```
IDispatch* CEx23cDoc::CreateAlarm(DATE time)
{
   AFX_MANAGE_STATE(AfxGetAppModuleState());
   TRACE("Entering CEx23cDoc::CreateAlarm, time = %f\n", time);
   // OLE уничтожает все существующие объекты CAlarm
   m_pAlarm = new CAlarm(time);
   return m_pAlarm->GetIDispatch(FALSE); // AddRef не используется
}
```

Мы предпочли, чтобы компонент создавал объект *CAlarm*, когда контроллер вызывает метод *CreateAlarm*. *CAlarm* — это класс компонента Automation, который мы сгенерировали при помощи Add Class Wizard. В нем *не предусмотрено* создание его объектов через COM, чем и обусловлено отсутствие для этого класса макроса *IMPLEMENT_OLECREATE* и фабрики классов. Функция *CreateAlarm* создает объект *CAlarm* и возвращает указатель на его интерфейс *IDispatch*. (Параметр *FALSE*, передаваемый в *CCmdTarget::GetIDispatch*, означает, что счетчик ссылок не увеличивается; счетчик ссылок на объект *CAlarm* уже установлен в 1 при создании объекта.)

Класс *CAlarm* объявлен в Alarm.h так:

```
#pragma once
// CAlarm command target
class CAlarm : public CCmdTarget
{
   DECLARE_DYNAMIC(CAlarm)
public:
   CAlarm(DATE time);
   virtual ~CAlarm();
```

```
   virtual void OnFinalRelease();
   DATE m_time;

protected:
   DECLARE_MESSAGE_MAP()
   DECLARE_DISPATCH_MAP()
   DECLARE_INTERFACE_MAP()
   void OnTimeChanged(void);

   enum
   {
      dispidTime = 1
   };
};
```

Обратите внимание на отсутствие макроса *DECLARE_DYNCREATE*.

В файле Alarm.cpp содержится карта диспетчеризации:

```
BEGIN_DISPATCH_MAP(CAlarm, CCmdTarget)
   DISP_PROPERTY_NOTIFY_ID(CAlarm, "Time",
      dispidTime, m_time, OnTimeChanged, VT_DATE)
END_DISPATCH_MAP()
```

Для чего нам класс *CAlarm*? Вместо него мы могли бы добавить в класс *CEx23cDoc* свойство *AlarmTime*, но тогда для включения и отключения подачи сигнала будильником понадобилось бы другое свойство или метод. Чего мы действительно добились, введя класс *CAlarm*, так это поддержки *набора* сигналов.

Для реализации набора сигналов мы могли бы написать еще один класс — *CAlarms* с методами *Add*, *Remove* и *Item*. Назначение первых двух понятно (добавление и удаление отдельных элементов набора), а *Item* возвращает указатель *IDispatch* элемента набора, задаваемого индексом, числом или чем-то еще. Мы также могли бы реализовать свойство *Count*, доступное только для чтения и возвращающее число элементов. В классе документа, в котором содержится набор, был бы метод *Alarms* с необязательным параметром типа *VARIANT*. Если этого параметра нет, метод возвращает указатель на *IDispatch* для набора, а если в параметре задан индекс — указатель на *IDispatch* для выбранного сигнала.

Примечание Пожелай мы, чтобы набор поддерживал характерный для VBA синтаксис типа «For Each», нам пришлось бы проделать кое-что еще. К классу *CAlarms* надо было бы добавить интерфейс *IEnumVARIANT* и реализовать функцию-член *Next* этого интерфейса для прохода по элементам набора. Затем ввести в класс *CAlarms* метод *_NewEnum*, возвращающий указатель на интерфейс *IEnumVARIANT*. А чтобы набор был универсальным, нужно было бы разрешить создание отдельных *объектов-перечислителей* (enumerator object) (с интерфейсом *IEnumVARIANT*) и реализовать другие функции *IEnumVariant*: *Skip*, *Reset* и *Clone*.

Свойство Figures интересно тем, что допускает индексацию. Оно представляет четыре числа из римских цифр на циферблате часов: XII, III, VI и IX. Свойство

реализовано как массив *CString*, так что применение римских цифр вполне допустимо. Объявление в Ex23cDoc.h выглядит так:

```
public:
   CString m_strFigure[4];
```

А функции *GetFigure* и *SetFigure* в Ex23cDoc.cpp реализованы так:

```
VARIANT CEx23cDoc::GetFigure(SHORT n)
{
   AFX_MANAGE_STATE(AfxGetAppModuleState());
   TRACE("Entering CEx23cDoc::GetFigure - n = %d m_strFigure[n] = %s\n",
         n, m_strFigure[n]);
   return COleVariant(m_strFigure[n]).Detach();
}
void CEx23cDoc::SetFigure(SHORT n, VARIANT FAR& newVal)
{
   AFX_MANAGE_STATE(AfxGetAppModuleState());
   TRACE("Entering CEx23cDoc::SetFigure - n = %d, vt = %d\n", n,
         newVal.vt);
   COleVariant vaTemp;
   vaTemp.ChangeType(VT_BSTR, (COleVariant*) &newVal);
   m_strFigure[n] = vaTemp.bstrVal; // преобразует двухбайтовую строку в однобайтовую
   SetModifiedFlag();
}
```

С этими функциями связан макрос *DISP_PROPERTY_PARAM* из карты диспетчеризации класса *CEx23cDoc*. Первый параметр функций — индекс, заданный последним параметром макроса как тип короткое целое. Индексы свойств не обязательно должны быть целыми числами и могут состоять из нескольких компонентов (например, номеров строки и столбца). Вызов *ChangeType* в *SetFigure* обязателен, иначе контроллер передаст вместо строк числа.

Мы только что рассмотрели свойства-наборы и индексируемые свойства. Чем они различаются? Контроллер не может добавлять и удалять элементы индексируемого свойства, но может делать это в наборе.

Какая функция рисует циферблат? Как вы, наверное, и ожидали, этим занимается функция *OnDraw* класса «вид». Функция использует *GetDocument*, чтобы получить указатель на документ, а затем обращается к переменным-членам и функциям-членам, соответствующим свойствам и методам.

И, наконец, код макроса Excel:

```
Dim Clock As Object
Dim Alarm As Object

Sub LoadClock()
   Set Clock = CreateObject("Ex23c.Document")
   Range("A3").Select
   n = 0
   Do Until n = 4
      Clock.figure(n) = Selection.Value
      Selection.Offset(0, 1).Range("A1").Select
```

```
        n = n + 1
    Loop
    RefreshClock
    Clock.ShowWin
End Sub

Sub RefreshClock()
    Clock.Time = Now()
    Clock.RefreshWin
End Sub

Sub CreateAlarm()
    Range("E3").Select
    Set Alarm = Clock.CreateAlarm(Selection.Value)
    RefreshClock
End Sub

Sub UnloadClock()
    Set Clock = Nothing
End Sub
```

Обратите внимание на оператор *Set Alarm* в макросе *CreateAlarm*. Он вызывает метод *CreateAlarm*, чтобы получить указатель на *IDispatch*, сохраняемый в объектной переменной. Запустив макрос повторно, вы зададите новый будильник, но при этом уничтожите старый, так как его счетчик ссылок обнулится.

Внимание! Вы видели модальное диалоговое окно в DLL (пример Ex23b) и основное окно-рамку в EXE-модуле (пример Ex23c). Будьте осторожны с модальными диалоговыми окнами в EXE. Диалоговое окно About, вызываемое прямо из компонентной программы, — вещь нормальная, но активизация модального диалогового окна из функции-метода внешнего компонента — идея не из лучших. Проблема в том, что, когда на экране открыто модальное диалоговое окно, пользователь, может вновь переключиться на клиентскую программу-контроллер. MFC-контроллеры реагируют на это, открывая окно с сообщением о занятости сервера (Server Busy). Excel поступает аналогично, но перед этим зачем-то выжидает 30 секунд, что часто сбивает с толку.

Пример Ex23d: клиент Automation

Итак, вы познакомились с примерами компонентов Automation. Теперь перейдем к примеру клиентской программы Automation на C++, которая запускает все эти компоненты, а также управляет работой Microsoft Excel. Программа Ex23d изначально сгенерирована MFC Application Wizard, но без установки флажков, связанных с COM. Проще добавить относящийся к COM код вручную, чем удалять код, связанный с компонентом. Если же вы все-таки решите генерировать подобный контроллер Automation с помощью MFC Application Wizard, добавьте в конец StdAfx.h строку:

```
#include <afxdisp.h>
```

а в начало *InitInstance* приложения — вызов:

```
AfxOleInit();
```

Чтобы подготовить исполняемый файл Ex23d, откройте и соберите решение \vcppnet\Ex23d\Ex23d.sln. Запустив программу под управлением отладчика, вы увидите стандартное SDI-приложение, с такой структурой меню (рис. 23-5):

File	Edit	Bank Comp	DLL Comp	Clock Comp	Excel Comp	View	Help
		Load	Load	Load	Load		
		Test	Get Data	Create Alarm	Execute		
		Unload	Unload	Refresh Time			
				Unload			

Рис. 23-5. *Структура меню SDI-приложения Ex23d*

Собрав и зарегистрировав все компоненты, можете протестировать их с помощью Ex23d. Заметьте: DLL-компоненты не обязательно размещать в каталоге \Winnt\System32, так как Windows отыскивает их по информации в реестре. Для некоторых компонентов, чтобы убедиться в правильности результатов теста, придется вывести окно Debug. Программа Ex23d достаточно модульна. Команды меню и события, связанные с обновлением пользовательского интерфейса, направляются в класс «вид». У каждого компонентного объекта свой класс C++ контроллера и внедренная переменная-член в Ex23dView.h. Мы рассмотрим каждую часть программы по отдельности, но сначала разберемся с библиотеками типов.

Библиотеки типов и IDL-файлы

Мы уже говорили, что библиотеки типов в MFC-реализации *IDispatch* не нужны, но Visual Studio .NET все равно «втихую» генерирует их для всех создаваемых компонентов. В чем их польза? Дело в том, что VBA они могут понадобиться для просмотра методов и свойств компонента и для ускорения доступа к методам и свойствам в процессе *раннего связывания* (early binding), о котором мы поговорим чуть ниже. Но мы ведь создаем здесь клиентскую программу на C++, а не на VBA. Оказывается, Add Class Wizard способен — по информации из библиотеки типов компонента — генерировать код C++, позволяющий контроллеру «управлять» компонентом Automation.

Примечание MFC Application Wizard инициализирует IDL-файл (Interface Definition Language — язык описания интерфейсов) при создании проекта. Мастера Add Property Wizard и Add Method Wizard обновляют этот файл всякий раз, когда вы генерируете новый класс компонента Automation или добавляете свойства и методы к существующему классу.

Когда вы добавляли свойства и методы к классам компонентов, Add Method Wizard и Add Property Wizard обновляли IDL-файл проекта. Это текстовый файл,

описывающий компонент на языке описания объектов. Ниже приведен IDL-файл для банковского компонента (если вы сгенерируете этот проект с помощью MFC Application Wizard, GUID в вашем файле будет другим).

```
// Ex23a.idl : type library source for Ex23a.exe
// This file will be processed by the MIDL compiler to produce the
// type library (Ex23a.tlb).
#include "olectl.h"
[ uuid(60BCA7D2-14D1-4832-A278-50670CD9975E), version(1.0) ]
library Ex23a
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

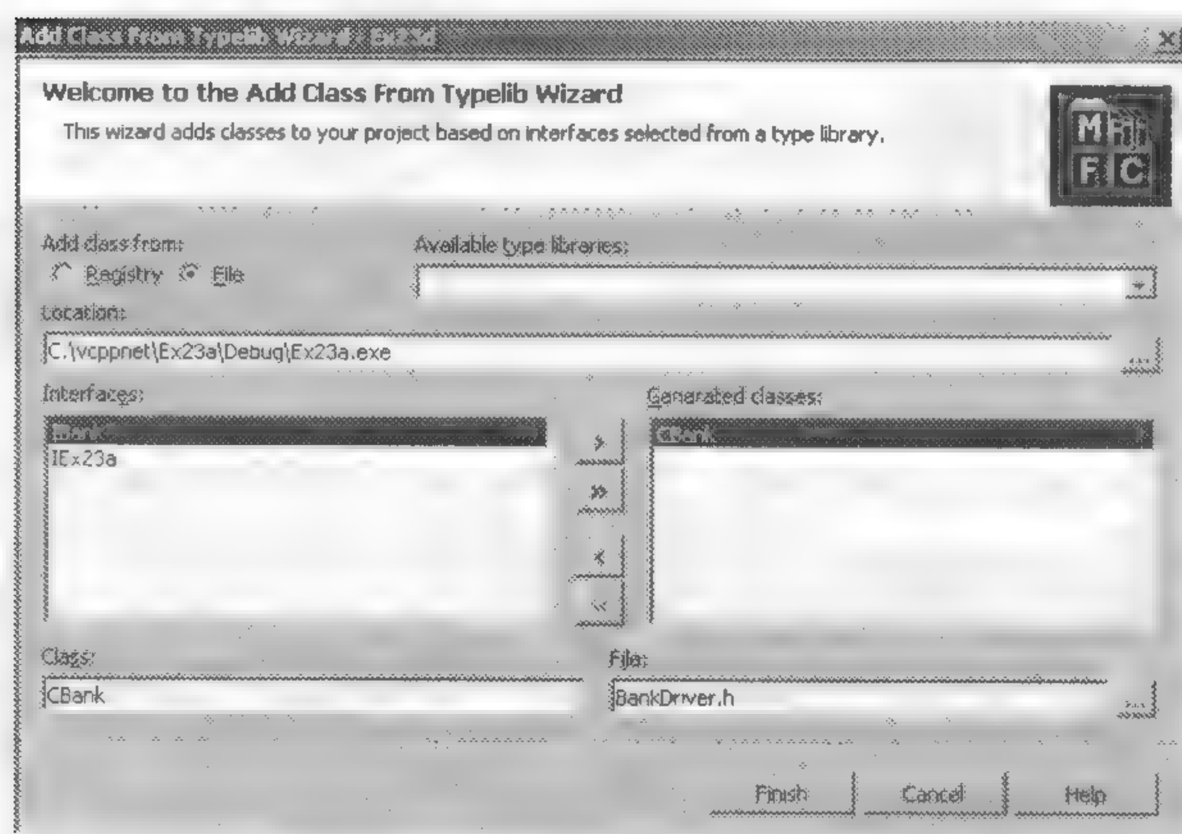
   // Primary dispatch interface for CEx23aDoc
   [ uuid(1F013122-EA3D-414F-B58F-5A31A64EA5D5) ]
   dispinterface IEx23a
   {
      properties:
      methods:
   };
   // Class information for CEx23aDoc
   [ uuid(5EE5C98C-5CCF-46F4-9E95-17BC06237D8B) ]
   coclass Ex23a
   {
      [default] dispinterface IEx23a;
   };
   // Primary dispatch interface for Bank
   [ uuid(8BAD2B0C-62CC-4952-811C-C736DA06858E) ]
   dispinterface IBank
   {
      properties:
      [id(3), helpstring("property Balance")] DOUBLE Balance;
      methods:
      [id(1), helpstring("method Withdrawal")]
         DOUBLE Withdrawal(DOUBLE dAmount);
      [id(2), helpstring("method Deposit")]
         void Deposit(DOUBLE dAmount);
   };
   // Class information for Bank
   [ uuid(3EC6FA59-9F9F-4619-9F62-BA5FE37176F0) ]
   coclass Bank
   {
      [default] dispinterface IBank;
   };
};
```

ODL-файл содержит уникальный GUID-идентификатор библиотеки типов 60BCA7D2-14D1-4832-A278-50670CD9975E и полностью описывает свойства и методы в *диспетчерском интерфейсе* (dispinterface) *IBank*. Кроме того, для dispinterface задан свой GUID (8BAD2B0C-62CC-4952-811C-C736DA06858E), это тот же идентифика-

тор, что и в таблице интерфейсов класса *CBank*. Смысл этого GUID разъясняется в разделе «Раннее связывание в VBA» в конце главы. Для загрузки компонента VBA применяет идентификатор класса (3EC6FA59-9F9F-4619-9F62-BA5FE37176F0).

В любом случае Visual Studio .NET при сборке проекта *компонента* вызывает утилиту MIDL, которая считывает IDL-файл и генерирует двоичный TLB-файл в каталоге Debug или Release проекта. Теперь, когда вы создаете на C++ *клиентскую программу*, из TLB-файла проекта компонента можно сгенерировать с помощью Add Class Wizard *управляющий* (driver) класс.

Чтобы все это проделать, щелкните кнопку Add Class в меню Project и в открывшемся окне выберите в списке шаблон MFC Class From TypeLib. Укажите TLB-файл проекта компонента, и Add Class Wizard откроет диалоговое окно вроде этого:



IBank — это имя disp-интерфейса, заданное в IDL-файле. Можете, если хотите, сохранить это имя класса, а также задать имена заголовочного и CPP-файлов. Если в библиотеке типов не один интерфейс, можно выбрать сразу несколько интерфейсов. Сгенерированный класс контроллера показан в следующем разделе.

Класс контроллера для Ex23a.exe

Мастер Add Class From Typelib Wizard сгенерировал класс *IBank* (производный от *COleDispatchDriver*) (см. листинг ниже). Присмотритесь к реализации функций-членов. Обратите внимание на первый параметр в вызовах функций *GetProperty*, *SetProperty* и *InvokeHelper*. Это жестко заданные DISPID-идентификаторы свойств или методов компонента в порядке, определенном в карте диспетчеризации компонента.

BankDriver.h

```
class CBank : public COleDispatchDriver
{
public:
    CBank(){} // Calls COleDispatchDriver default constructor
    CBank(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
    CBank(const CBank& dispatchSrc) : COleDispatchDriver(dispatchSrc) {}
    // Attributes
```

```
public:
    // operations
public:
    // IBank methods
public:
    double Withdrawal(double dAmount)
    {
        double result;
        static BYTE parms[] = VTS_R8 ;
        InvokeHelper(0x1, DISPATCH_METHOD, VT_R8, (void*)&result,
                     parms, dAmount);
        return result;
    }
    void Deposit(double dAmount)
    {
        static BYTE parms[] = VTS_R8 ;
        InvokeHelper(0x2, DISPATCH_METHOD, VT_EMPTY, NULL,
                     parms, dAmount);
    }
    // IBank properties
public:
    double GetBalance()
    {
        double result;
        GetProperty(0x3, VT_R8, (void*)&result);
        return result;
    }
    void SetBalance(double propVal)
    {
        SetProperty(0x3, VT_R8, propVal);
    }
};
```

В классе *CEx23dView* есть переменная-член *m_bank* класса *IBank*. Функции-члены *CEx23dView* для компонента Ex23a.Bank приведены ниже. Они вызываются при выборе команд из основного меню контроллера. В частности, представляет интерес функция *OnBankoleLoad*. Функция *COleDispatchDriver::CreateDispatch* загружает программу компонента вызовом *CoGetClassObject* и *IClassFactory::CreateInstance*. Затем она вызывает *QueryInterface*, чтобы получить указатель на *IDispatch*, сохраняемый в переменной-члене *m_lpDispatch*. Функция *COleDispatchDriver::ReleaseDispatch*, вызываемая из *OnBankoleUnload*, освобождает этот указатель вызовом *Release*.

```
void CEx23dView::OnBankoleLoad()
{
    if(!m_bank.CreateDispatch("Ex23a.Bank")) {
        AfxMessageBox("Ex23a.Bank component not found");
        return;
    }
}
```

```
void CEx23dView::OnUpdateBankoleLoad(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_bank.m_lpDispatch == NULL);
}
void CEx23dView::OnBankoleTest()
{
    m_bank.Deposit(20.0);
    m_bank.Withdrawal(15.0);
    TRACE("new balance = %f\n", m_bank.GetBalance());
}
void CEx23dView::OnUpdateBankoleTest(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_bank.m_lpDispatch != NULL);
}
void CEx23dView::OnBankoleUnload()
{
    m_bank.ReleaseDispatch();
}
void CEx23dView::OnUpdateBankoleUnload(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_bank.m_lpDispatch != NULL);
}
```

Класс контроллера для Ex23b.dll

На рис. 23-9 показан заголовочный файл класса, сгенерированный мастером Add Class From Typelib Wizard.

AutoDriver.h

```
// Machine generated IDispatch wrapper class(es) created with
// Add Class from Typelib Wizard

// CEx23bAuto wrapper class
class CEx23bAuto : public COleDispatchDriver
{
public:
    CEx23bAuto(){} // Calls COleDispatchDriver default constructor
    CEx23bAuto(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
    CEx23bAuto(const CEx23bAuto& dispatchSrc) :
        COleDispatchDriver(dispatchSrc) {}

    // Attributes
public:
    // Operations
public:
    // IEx23bAuto methods
public:
    BOOL DisplayDialog()
    {
        BOOL result;
```

```
        InvokeHelper(0x3, DISPATCH_METHOD, VT_BOOL,
                  (void*)&result, NULL);
        return result;
    }
    // IEx23bAuto properties
public:
    long GetLongData()
    {
        long result;
        GetProperty(0x1, VT_I4, (void*)&result);
        return result;
    }
    void SetLongData(long propVal)
    {
        SetProperty(0x1, VT_I4, propVal);
    }
    VARIANT GetTextData()
    {
        VARIANT result;
        GetProperty(0x2, VT_VARIANT, (void*)&result);
        return result;
    }
    void SetTextData(const VARIANT& propVal)
    {
        SetProperty(0x2, VT_VARIANT, &propVal);
    }
};
```

Заметьте: для каждого свойства нужны отдельные функции *Get* и *Set*, даже если какое-то свойство представлено в компоненте как переменная-член.

Заголовочный файл класса «вид» задает переменную-член *m_auto* класса *CEx23b-Auto*. Ниже приведены функции-члены из Ex23dView.cpp — обработчики команд из Ex23dView.cpp.

```
void CEx23dView::OnDlloleGetdata()
{
    m_auto.DisplayDialog();
    COleVariant vaData = m_auto.GetTextData();
    ASSERT(vaData.vt == VT_BSTR);
    CString strTextData(vaData.bstrVal);
    long lData = m_auto.GetLongData();
    TRACE("CEx23dView::OnDlloleGetdata - long = %ld, text = %s\n",
          lData, strTextData);
}
void CEx23dView::OnUpdateDlloleGetdata(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.m_lpDispatch != NULL);
}
void CEx23dView::OnDlloleLoad()
{
```

```
    if(!m_auto.CreateDispatch("Ex23b.Ex23bAuto")) {
        AfxMessageBox("Ex23b.Ex23bAuto component not found");
        return;
    }
    COleVariant va("test");
    m_auto.SetTextData(va); // testing
    m_auto.SetLongData(79); // testing
    // verify dispatch interface
    // {125FECB2-734D-49FD-95C7-FE44B77FDE2C}
    static const IID IID_IEx23bAuto =
        { 0x125FECB2, 0x734D, 0x49FD, { 0x95, 0xC7, 0xFE,
            0x44, 0xB7, 0x7F, 0xDE, 0x2C } };
    LPDISPATCH p;
    HRESULT hr = m_auto.m_lpDispatch->QueryInterface(IID_IEx23bAuto,
                                                     (void**) &p);
    TRACE("OnDlloleLoad - QueryInterface result = %x\n", hr);
    p->Release();
}
void CEx23dView::OnUpdateDlloleLoad(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.m_lpDispatch == NULL);
}
void CEx23dView::OnDlloleUnload()
{
    m_auto.ReleaseDispatch();
}
void CEx23dView::OnUpdateDlloleUnload(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.m_lpDispatch != NULL);
}
```

Класс контроллера для Ex23c.exe

Ниже показаны заголовки классов *CEx23c* и *IAlarm*, управляющих компонентом Automation Ex23c.

ClockDriver.h

```
// Machine generated IDispatch wrapper class(es) created with
// Add Class from Typelib Wizard

// CEx23c wrapper class
class CEx23c : public COleDispatchDriver
{
public:
    CEx23c(){} // Calls COleDispatchDriver default constructor
    CEx23c(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
    CEx23c(const CEx23c& dispatchSrc) :
        COleDispatchDriver(dispatchSrc) {}
```

```
[illegible]
```

CAlarm.h

```
class CAlarm : public COleDispatchDriver
{
public:
   CAlarm(){} // Calls COleDispatchDriver default constructor
   CAlarm(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
   CAlarm(const CAlarm& dispatchSrc) :
      COleDispatchDriver(dispatchSrc) {}

   // Attributes
public:
   // Operations
public:
   // IAlarm methods
public:
   // IAlarm properties
public:
   DATE GetTime()
   {
      DATE result;
      GetProperty(0x1, VT_DATE, (void*)&result);
      return result;
   }
   void SetTime(DATE propVal)
   {
      SetProperty(0x1, VT_DATE, propVal);
   }
};
```

Особый интерес для нас представляет функция-член *CEx23c::CreateAlarm* из ClockDriver.cpp. Ее можно вызывать только после создания объекта (документа) «будильник». Она заставляет компонент Ex23c создать объект «сигнал» и возвращает указатель на *IDispatch* со счетчиком ссылок, равным 1. Функция *COleDispatchDriver::AttachDispatch* подсоединяет этот указатель к клиентскому объекту *m_alarm*, но если у этого объекта уже есть диспетчерский указатель, то он освобождается. Именно поэтому вы видите в окне отладки, что старый экземпляр Ex23c завершается сразу же после запроса нового. Протестировать подобное поведение с контроллером Excel нельзя, так как Ex23d отключает в своем меню команду Load после запуска компонента «будильник».

В классе «вид» две переменных-члена: *m_clock* и *m_alarm*. Вот как выглядят обработчики команд из этого класса:

```
void CEx23dView::OnClockoleCreatealarm()
{
   CAlarmDialog dlg;
   if (dlg.DoModal() == IDOK) {
      COleDateTime dt(2002, 12, 23, dlg.m_nHours, dlg.m_nMinutes,
                   dlg.m_nSeconds);
      LPDISPATCH pAlarm = m_clock.CreateAlarm(dt);
```

```
        m_alarm.AttachDispatch(pAlarm); // releases prior object!
        m_clock.RefreshWin();
    }
}
void CEx23dView::OnUpdateClockoleCreatealarm(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.m_lpDispatch != NULL);
}
void CEx23dView::OnClockoleLoad()
{
    if(!m_clock.CreateDispatch("Ex23c.Document")) {
        AfxMessageBox("Ex23c.Document component not found");
        return;
    }
    m_clock.put_Figure(0, COleVariant("XII"));
    m_clock.put_Figure(1, COleVariant("III"));
    m_clock.put_Figure(2, COleVariant("VI"));
    m_clock.put_Figure(3, COleVariant("IX"));
    OnClockoleRefreshtime();
    m_clock.ShowWin();
}
void CEx23dView::OnUpdateClockoleLoad(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.m_lpDispatch == NULL);
}
void CEx23dView::OnClockoleRefreshtime()
{
    COleDateTime now = COleDateTime::GetCurrentTime();
    m_clock.SetTime(now);
    m_clock.RefreshWin();
}
void CEx23dView::OnUpdateClockoleRefreshtime(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.m_lpDispatch != NULL);
}
void CEx23dView::OnClockoleUnload()
{
    m_clock.ReleaseDispatch();
}
void CEx23dView::OnUpdateClockoleUnload(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.m_lpDispatch != NULL);
}
```

Управление приложением Microsoft Excel

Программа Ex23d содержит код, который, загрузив Excel и создав рабочую книгу, читает и изменяет содержимое ячеек активной таблицы. Управление Excel выполняется так же, как и управление MFC-компонентом Automation, но есть ряд особенностей, в которых вам нужно разобраться.

Изучая Excel VBA, вы обнаружите, что в программе можно задействовать свыше сотни различных «объектов». Все они доступны через Automation, но при написании котроллера автоматизации на MFC надо знать о свойствах и методах этих объектов. В идеале у каждого объекта с функциями-членами, вызываемыми по диспетчерским идентификаторам, должен быть свой класс C++.

У Excel есть собственная библиотека типов, зарегистрированная в реестре. Add Class From Typelib Wizard способен считывать этот файл (так же, как и TLB-файлы) и создавать на его основе классы C++ контроллеров для отдельных объектов Excel. Поэтому имеет смысл отобрать нужные объекты и свести полученные классы в отдельный набор файлов (рис. 23-6).

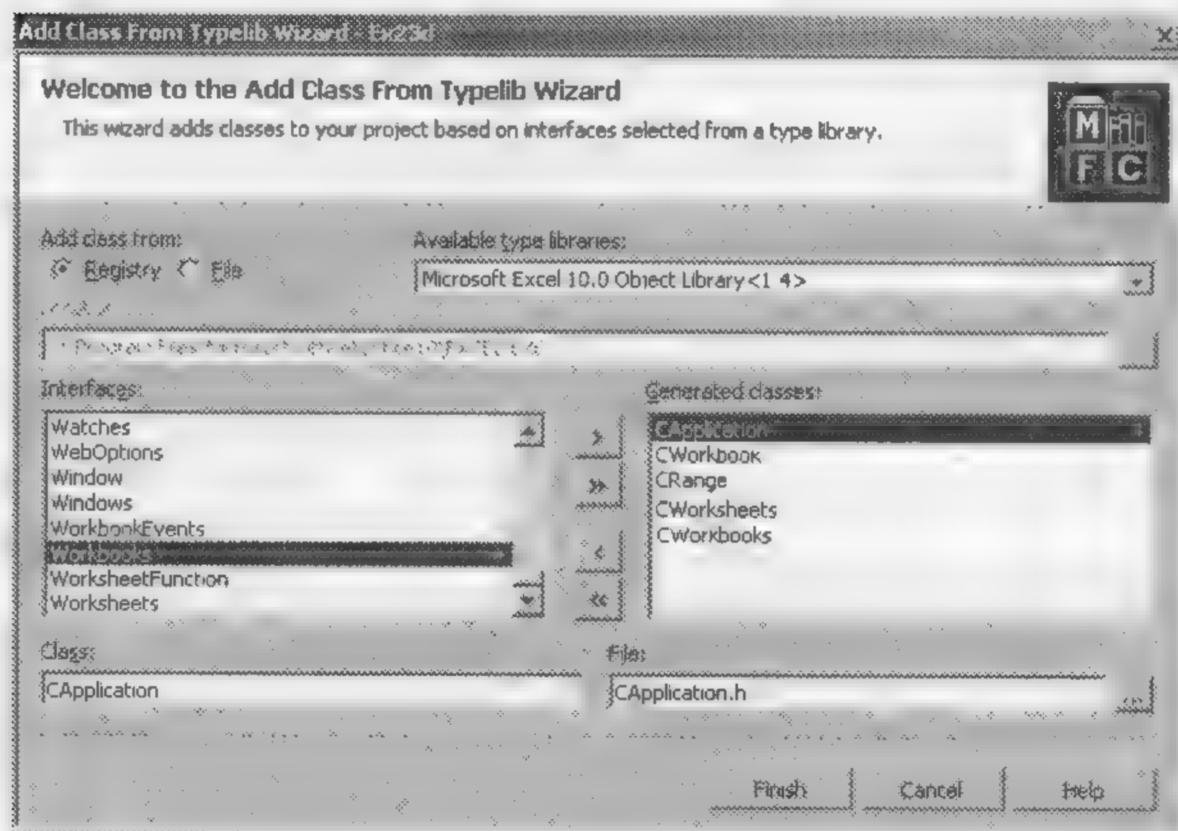


Рис. 23-6. *Add Class From Typelib Wizard способен создавать классы C++ для объектов Excel, перечисленных в библиотеке типов Excel*

Не исключено, что сгенерированный код придется редактировать вручную в соответствии со своими потребностями. Рассмотрим это на примере. Сформировав с помощью Add Class From Typelib Wizard класс контроллера для объекта Worksheet, вы получите такую функцию-член *get_Range*:

```
LPDISPATCH get_Range(VARIANT Cell1, VARIANT Cell2)
{
   LPDISPATCH result;
   static BYTE parms[] = VTS_VARIANT VTS_VARIANT ;
   InvokeHelper(0xc5, DISPATCH_PROPERTYGET, VT_DISPATCH,
             (void*)&result, parms, &Cell1, &Cell2);
   return result;
}
```

Как известно из документации по Excel VBA, этот метод можно вызывать как для одной ячейки (один параметр), так и для прямоугольной области, определяемой двумя ячейками (два параметра). Вспомните: в вызове *InvokeHelper* можно не передавать необязательные параметры. В данном случае можно добавить вторую перегруженную функцию *get_Range* с одним параметром-ячейкой:

```
LPDISPATCH get_Range( VARIANT Cell1) // Добавлено
{
```

```
    LPDISPATCH result;
    static BYTE parms[] = VTS_VARIANT;
    InvokeHelper(0xc5, DISPATCH_PROPERTYGET, VT_DISPATCH,
            (void*)&result, parms, &Cell1);
    return result;
}
```

Какие же функции подправить? Да те, что вы решили использовать в своей программе. Прочтите руководство по Excel VBA, чтобы выяснить обязательные параметры и возвращаемые значения. Может быть, кто-то скоро напишет набор контроллерных классов на C++ для Excel.

Объекты Excel, используемые программой Ex23d, и соответствующие им классы описаны в таблице. (Код содержится в файлах CApplication.h, CRange.h, CWorksheet.h, CWorksheets.h и CWorkbooks.h.)

Класс	Переменная-член класса «вид»
CApplication	*m_app*
CRange	*m_range[5]*
CWorksheet	*m_worksheet*
CWorkbooks	*m_workbooks*
CWorksheets	*m_worksheets*

Команду Load меню Excel Comp в примере обрабатывает функция-член *OnExceloleLoad* класса «вид». Эта функция должна работать, даже если Excel уже запущен пользователем. COM-функция *GetActiveObject* пытается возвратить указатель на *IUnknown* для Excel. *GetActiveObject* требует идентификатор класса, поэтому мы сначала вызываем *CLSIDFromProgID*. Если вызов *GetActiveObject* успешен, вызывается *QueryInterface*, чтобы получить указатель на *IDispatch*, который передается контроллерному объекту *m_app* класса *Application*. Если же вызов *GetActiveObject* закончился неудачей, вызывается *COleDispatchDriver::CreateDispatch*, как было с другими компонентами.

```
void CEx23dView::OnExceloleLoad()
{   // если Excel уже запущен, подключаемся к нему; нет - запускаем его
    LPDISPATCH pDisp;
    LPUNKNOWN pUnk;
    CLSID clsid;
    TRACE("Entering CEx23dView::OnExcelLoad\n");
    BeginWaitCursor();
    // Используем Excel.Application.9 в Office 2000
    // Используем Excel.Application.10 в Office XP
    ::CLSIDFromProgID(L"Excel.Application.10", &clsid); // из  реестра
    if(::GetActiveObject(clsid, NULL, &pUnk) == S_OK) {
        VERIFY(pUnk->QueryInterface(IID_IDispatch,
                (void**) &pDisp) == S_OK);
        m_app.AttachDispatch(pDisp);
        pUnk->Release();
        TRACE(" attach complete\n");
    }
    else {
```

```
        if(!m_app.CreateDispatch("Excel.Application.10")) {
            AfxMessageBox("Microsoft Excel program not found");
        }
        TRACE(" create complete\n");
    }
    EndWaitCursor();
}
```

Главная задача *OnExceloleExecute* — функции-обработчика команды Execute в меню Excel Comp — найти основное окно Excel и сделать его активным (вывести на передний план). Для этого придется писать свой Windows-код — соответствующего метода у Excel нет. Мы должны создать и рабочую книгу, если на данный момент ни одна не открыта.

Значения, возвращаемые методами, надо тщательно отслеживать. Например, метод *Add* набора *Workbooks* возвращает указатель на *IDispatch* для объекта *Workbook* и, естественно, увеличивает счетчик ссылок. Если б мы сгенерировали класс для *Workbook*, то могли бы вызвать *AttachDispatch*, чтобы при уничтожении этого объекта вызывалась *Release*. Но поскольку класс *Workbook* нам не нужен, мы сами освобождаем указатель в конце функции. Если же вы забудете освободить указатели, отладочная версия MFC сообщит об утечке памяти.

В остальной части функции *OnExceloleExecute* осуществляется доступ к ячейкам рабочей таблицы. Здесь вы убедитесь, насколько просто считывать и задавать числа, даты, строки и формулы. Код C++ очень похож на VBA-код, который можно было бы написать для той же цели.

```
void CEx23dView::OnExceloleExecute()
{
    LPDISPATCH pRange, pWorkbooks;
    CWnd* pWnd = CWnd::FindWindow("XLMAIN", NULL);
    if (pWnd != NULL) {
        TRACE("Excel window found\n");
        pWnd->ShowWindow(SW_SHOWNORMAL);
        pWnd->UpdateWindow();
        pWnd->BringWindowToTop();
    }
    m_app.put_SheetsInNewWorkbook(1);

    VERIFY(pWorkbooks = m_app.get_Workbooks());
    m_workbooks.AttachDispatch(pWorkbooks);

    LPDISPATCH pWorkbook = NULL;
    if (m_workbooks.get_Count() == 0) {
        // Add возвращает указатель на Workbook,
        // но у нас нет класса Workbook
        pWorkbook = m_workbooks.Add(COleVariant((short) 0)); // Сохраняем указатель,
                                                              // чтобы освободить позже
    }
    LPDISPATCH pWorksheets = m_app.get_Worksheets();
    ASSERT(pWorksheets != NULL);
    m_worksheets.AttachDispatch(pWorksheets);
```

```
LPDISPATCH pWorksheet = m_worksheets.get_Item(COleVariant((short) 1));

m_worksheet.AttachDispatch(pWorksheet);
m_worksheet.Select(COleVariant((short) 0));

VERIFY(pRange = m_worksheet.get_Range(COleVariant("A1"),
                                      COleVariant("A1")));
m_range[0].AttachDispatch(pRange);

VERIFY(pRange = m_worksheet.get_Range(COleVariant("A2"),
                                      COleVariant("A2")));
m_range[1].AttachDispatch(pRange);

VERIFY(pRange = m_worksheet.get_Range(COleVariant("A3"),
                                      COleVariant("A3")));
m_range[2].AttachDispatch(pRange);

VERIFY(pRange = m_worksheet.get_Range(COleVariant("A3"),
                                      COleVariant("C5")));
m_range[3].AttachDispatch(pRange);

VERIFY(pRange = m_worksheet.get_Range(COleVariant("A6"),
                                      COleVariant("A6")));
m_range[4].AttachDispatch(pRange);

m_range[4].put_Value(COleVariant(COleDateTime(2002, 4, 24,
                                              15, 47, 8)));
// Получаем сохраненную дату и выводим ее как строку
COleVariant vaTimeDate = m_range[4].get_Value();
TRACE("returned date type = %d\n", vaTimeDate.vt);
COleVariant vaTemp;
vaTemp.ChangeType(VT_BSTR, &vaTimeDate);
CString str(vaTemp.bstrVal);
TRACE("date = %s\n", (const char*) str);

m_range[0].put_Value(COleVariant("test string"));

COleVariant vaResult0 = m_range[0].get_Value();
if (vaResult0.vt == VT_BSTR) {
   CString str(vaResult0.bstrVal);
   TRACE("vaResult0 = %s\n", (const char*) str);
}
m_range[1].put_Value(COleVariant(3.14159));

COleVariant vaResult1 = m_range[1].get_Value();
if (vaResult1.vt == VT_R8) {
   TRACE("vaResult1 = %f\n", vaResult1.dblVal);
}
m_range[2].put_Formula(COleVariant("=$A2*2.0"));

COleVariant vaResult2 = m_range[2].get_Value();
```

```
    if (vaResult2.vt == VT_R8) {
        TRACE("vaResult2 = %f\n", vaResult2.dblVal);
    }
    COleVariant vaResult2a = m_range[2].get_Formula();
    if (vaResult2a.vt == VT_BSTR) {
        CString str(vaResult2a.bstrVal);
        TRACE("vaResult2a = %s\n", (const char*) str);
    }
    m_range[3].FillRight();
    m_range[3].FillDown();
    // очистка
    if (pWorkbook != NULL) {
        pWorkbook->Release();
    }
}
```

Пример Ex23e: клиент Automation

Эта программа использует директиву *#import* для генерации smart-указателей. Ведет себя она практически так же, как и Ex23d, разве что не запускает Excel. Директиву *#import* мы поместим в файл StdAfx.h, чтобы компилятор не создавал управляющие классы несколько раз. Вот какой код нужно добавить:

```
#include <afxdisp.h>
#import "..\Ex23a\Debug\Ex23a.tlb"  rename_namespace("BankDriv")
using namespace BankDriv;

#import "..\Ex23b\Debug\Ex23b.tlb"  rename_namespace("Ex23bDriv")
using namespace Ex23bDriv;

#import "..\Ex23c\Debug\Ex23c.tlb"  rename_namespace("ClockDriv")
using namespace ClockDriv;
```

Если в MFC Application Wizard установлен флажок ActiveX controls, мастер создаст вызов *AfxOleInit* в функции-члене *InitInstance* класса приложения (иначе это придется сделать вручную).

Заголовочный файл класса «вид» содержит такие встроенные smart-указатели:

```
IEx23bAutoPtr m_auto;
IBankPtr m_bank;
IEx23cPtr m_clock;
IAlarmPtr m_alarm;
```

А вот код обработчиков команд меню в классе «вид»:

```
void CEx23eView::OnBankoleLoad()
{
    if(m_bank.CreateInstance(__uuidof(Bank)) != S_OK) {
        AfxMessageBox("Bank component not found");
        return;
    }
}
```

```
void CEx23eView::OnUpdateBankoleLoad(CCmdUI *pCmdUI)
{
   pCmdUI->Enable(m_bank.GetInterfacePtr() == NULL);
}

void CEx23eView::OnBankoleTest()
{
   try {
      m_bank->Deposit(20.0);
      m_bank->Withdrawal(15.0);
      TRACE("new balance = %f\n", m_bank->GetBalance());
   } catch(_com_error& e) {
      AfxMessageBox(e.ErrorMessage());
   }
}
void CEx23eView::OnUpdateBankoleTest(CCmdUI *pCmdUI)
{
   pCmdUI->Enable(m_bank.GetInterfacePtr() != NULL);
}
void CEx23eView::OnBankoleUnload()
{
   m_bank.Release();
}
void CEx23eView::OnUpdateBankoleUnload(CCmdUI *pCmdUI)
{
   pCmdUI->Enable(m_bank.GetInterfacePtr() != NULL);
}
void CEx23eView::OnClockoleCreatealarm()
{
   CAlarmDlg dlg;
   try {
      if (dlg.DoModal() == IDOK) {
         COleDateTime dt(2001, 12, 23, dlg.m_nHours,
                         dlg.m_nMinutes, dlg.m_nSeconds);
         LPDISPATCH pAlarm = m_clock->CreateAlarm(dt);
         m_alarm.Attach((IAlarm*) pAlarm);  // releases prior object!
         m_clock->RefreshWin();
      }
   } catch(_com_error& e) {
      AfxMessageBox(e.ErrorMessage());
   }
}
void CEx23eView::OnUpdateClockoleCreatealarm(CCmdUI *pCmdUI)
{
   pCmdUI->Enable(m_clock.GetInterfacePtr() != NULL);
}
void CEx23eView::OnClockoleLoad()
{
   if(m_clock.CreateInstance(__uuidof(CEx23cDoc)) != S_OK) {
      AfxMessageBox("Clock component not found");
```

```
        return;
    }
    try {
        m_clock->PutFigure(0, COleVariant("XII"));
        m_clock->PutFigure(1, COleVariant("III"));
        m_clock->PutFigure(2, COleVariant("VI"));
        m_clock->PutFigure(3, COleVariant("IX"));
        OnClockoleRefreshtime();
        m_clock->ShowWin();
    } catch(_com_error& e) {
        AfxMessageBox(e.ErrorMessage());
    }
}
void CEx23eView::OnUpdateClockoleLoad(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.GetInterfacePtr() == NULL);
}
void CEx23eView::OnClockoleRefreshtime()
{
    COleDateTime now = COleDateTime::GetCurrentTime();
    try {
        m_clock->PutTime(now);
        m_clock->RefreshWin();
    } catch(_com_error& e) {
        AfxMessageBox(e.ErrorMessage());
    }
}
void CEx23eView::OnUpdateClockoleRefreshtime(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.GetInterfacePtr() != NULL);
}
void CEx23eView::OnClockoleUnload()
{
    m_clock.Release();
}
void CEx23eView::OnUpdateClockoleUnload(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_clock.GetInterfacePtr() != NULL);
}
void CEx23eView::OnDlloleGetdata()
{
    try {
        m_auto->DisplayDialog();
        COleVariant vaData = m_auto->GetTextData();
        ASSERT(vaData.vt == VT_BSTR);
        CString strTextData(vaData.bstrVal);
        long lData = m_auto->GetLongData();
        TRACE("CEx23dView::OnDlloleGetdata-long = %ld, text = %s\n",
              lData, strTextData);
    } catch(_com_error& e) {
```

```
        AfxMessageBox(e.ErrorMessage());
    }
}
void CEx23eView::OnUpdateDlloleGetdata(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.GetInterfacePtr() != NULL);
}
void CEx23eView::OnDlloleLoad()
{
    if(m_auto.CreateInstance(__uuidof(Ex23bAuto)) != S_OK) {
        AfxMessageBox("Ex23bAuto component not found");
        return;
    }
    IEx23bAuto* pEx23bAuto = 0;
    m_auto.QueryInterface(__uuidof(IEx23bAuto), (void**)&pEx23bAuto);
    if(pEx23bAuto) {
        pEx23bAuto->PutLongData(42);
        pEx23bAuto->Release();
    }
}
void CEx23eView::OnUpdateDlloleLoad(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.GetInterfacePtr() == NULL);
}
void CEx23eView::OnDlloleUnload()
{
    m_auto.Release();
}
void CEx23eView::OnUpdateDlloleUnload(CCmdUI *pCmdUI)
{
    pCmdUI->Enable(m_auto.GetInterfacePtr() != NULL);
}
```

Обратите внимание на блоки *try/catch* в функциях, которые работают с компонентами. Они особенно нужны для обработки ошибок, возникающих при прекращении работы программы компонента. В примере Ex23d этим занимается MFC-класс *COleDispatchDriver*.

Раннее связывание в VBA

Запуская компоненты Ex23a, Ex23b и Ex23c из Excel VBA, вы применяли метод *позднего связывания* (late binding). VBA, обращаясь к какому-то свойству или методу, обычно вызывает *IDispatch::GetIDsOfNames*, чтобы найти диспетчерский идентификатор по символьному имени. Это не только неэффективно — важнее то, что VBA не проверяет типы до фактического обращения к свойству или методу. Пусть VBA-программа пытается получить значение свойства, которое она считает числом, а компонент вместо этого возвращает строку. В этом случае VBA возвратит ошибку периода выполнения, лишь дойдя до оператора Property Get.

Однако *раннее связывание* (early binding) заставляет VBA предварительно обрабатывать исходный текст программы, преобразуя имена свойств и методов в

DISPID-идентификаторы до запуска программы компонента. При этом он проверяет типы свойств, возвращаемых значений и параметров методов, возвращая при необходимости ошибки компиляции. Но откуда VBA берет нужную информацию? Конечно же, из библиотеки типов компонента, благодаря которой VBA позволяет программисту просматривать описания свойств и методов сервера. VBA считывает библиотеку типов еще до загрузки компонентной программы.

Регистрация библиотеки типов

Вы уже видели, что Visual Studio .NET генерирует TLB-файл для каждого компонента. Чтобы VBA нашел библиотеку типов, ее адрес должен присутствовать в реестре. Записи в разделе TypeLib используются браузерами библиотек, а записи в разделе Interface служат для проверки типов в период выполнения и (в случае EXE-компонента) в процессе маршалинга для диспетчерского интерфейса.

Как компонент регистрирует свою библиотеку типов

Выполняясь в автономном режиме, EXE-компонент может вызывать функцию *AfxRegisterTypeLib* для внесения нужных записей в реестр. Например, так:

```
VERIFY(AfxOleRegisterTypeLib(AfxGetInstanceHandle(), theTypeLibGUID,
      "Ex23b.tlb"));
```

Здесь *theTypeLibGUID* — статическая переменная типа *GUID*.

```
// {A9515ACA-5B85-11D0-848F-00400526305B}
static const GUID theTypeLibGUID =
   { 0xa9515aca, 0x5b85, 0x11d0, { 0x84, 0x8f, 0x00, 0x40, 0x05, 0x26,
      0x30, 0x5b } };
```

Функция *AfxRegisterTypeLib* объявлена в файле afxwin.h, который требует определения макроса *_AFXDLL*. Таким образом, эту функцию нельзя использовать в обычной DLL, если только вы не скопируете ее код из исходных файлов MFC.

IDL-файл

Теперь самое время обратиться к IDL-файлу того же проекта:

```
// Ex23b.idl : type library source for Ex23b.dll
// This file will be processed by the MIDL compiler to produce the
// type library (Ex23b.tlb).

#include "olectl.h"
[ uuid(EE56DC40-B710-4543-8841-8D9C27ADA504), version(1.0) ]
library Ex23b
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");
   // Primary dispatch interface for Ex23bAuto

   [ uuid(125FECB2-734D-49FD-95C7-FE44B77FDE2C) ]
   dispinterface IEx23bAuto
   {
```

```
        properties:
        [id(1), helpstring("property LongData")] LONG LongData;
        [id(2), helpstring("property TextData")] VARIANT TextData;
        methods:
        [id(3), helpstring("method DisplayDialog")]
            VARIANT_BOOL DisplayDialog(void);
    };
    // Class information for Ex23bAuto
    [ uuid(BAF3D9ED-4518-43CA-B017-2EBA332CB618) ]
    coclass Ex23bAuto
    {
        [default] dispinterface IEx23bAuto;
    };
};
```

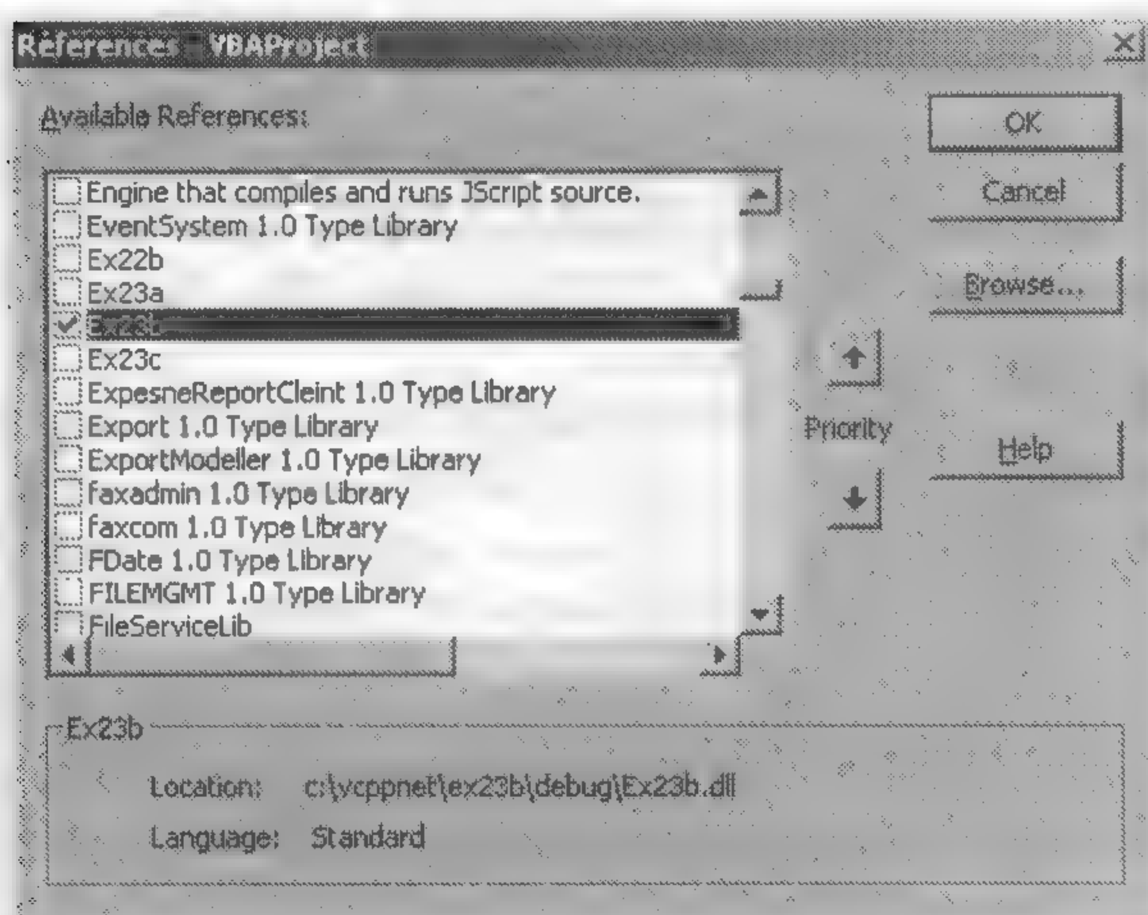
Как видите, между реестром, библиотекой типов, компонентом и клиентом VBA существует множество связей.

Примечание Полезная утилита OLEVIEW из состава Visual C++ позволяет просматривать зарегистрированные компоненты и их библиотеки типов.

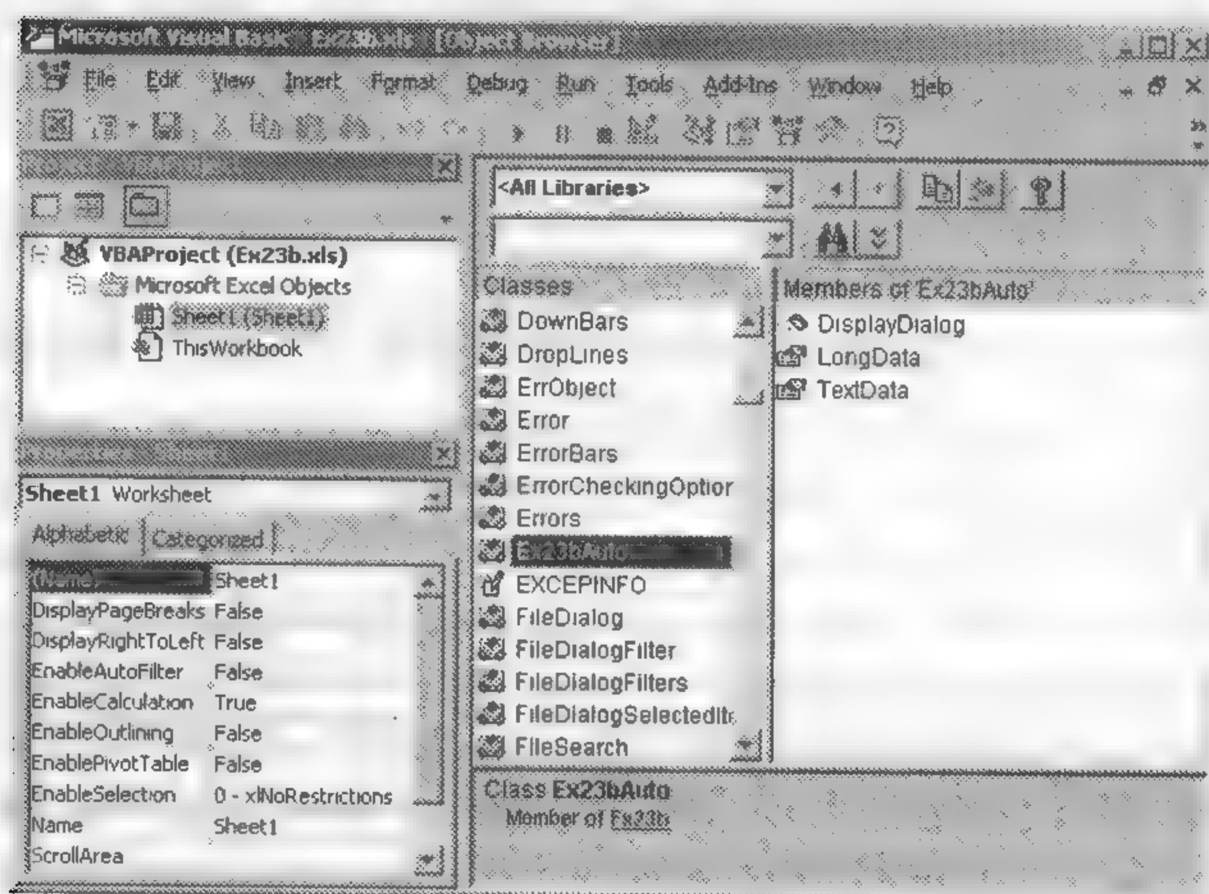
Использование библиотеки типов в Excel

Последовательность операций, выполняемых Excel для того, чтобы задействовать вашу библиотеку типов, такова.

1. При запуске Excel считывает из реестра раздел TypeLib и составляет список всех библиотек типов. Он загружает библиотеку типов VBA и библиотеку объектов Excel.
2. Пользователь (или автор рабочей книги) после запуска Excel, загрузки рабочей книги и переключения в окно редактора Visual Basic Editor выбирает в меню Tools команду References и отмечает флажок в строке Ex23b, как показано на рисунке. При сохранении рабочей книги информация о ссылках сохраняется вместе с ней.



3. Теперь пользователь Excel может просматривать свойства и методы Ex23b, выбрав в меню View команду Object Browser:



4. Чтобы ваша программа использовала библиотеку типов, просто замените строку

```
Dim DllComp As Object
```

на:

```
Dim DllComp As IEx23bAuto
```

Программа на VBA сразу завершится, если не найдет *IEx23bAuto* в списке ссылок.

5. Выполнив оператор CreateObject и загрузив компонент, VBA вызовет *QueryInterface* для *IID_IEx23bAuto*, определенного в реестре, библиотеке типов и таблице интерфейсов класса компонента. (На самом деле *IEx23bAuto* является интерфейсом *IDispatch*.) Это делается для большей безопасности. Если компонент не сможет предоставить этот интерфейс, программа на VBA завершится. Теоретически Excel мог бы применить для загрузки CLSID из библиотеки типов, но он использует CLSID из реестра, как и в режиме позднего связывания.

Зачем нужно раннее связывание

Вы, наверное, подумали, что раннее связывание ускорит работу компонента Automation. Но скорее всего вы не заметите этого из-за «узкого места» — вызовов *IDispatch::Invoke*. Типичное обращение клиента C++ к MFC-функции *Invoke* в компоненте C++ занимает около 0,5 мс, что совсем немало.

Возможность просмотра, предоставляемая библиотекой типов, вероятно, важнее, чем связывание на этапе компиляции. Если вы пишете контроллер на C++, библиотеку типов можно загружать через разные OLE-функции, в том числе *LoadTypeLib*, после чего она становится доступна через интерфейсы *ITypeLib* и *ITypeInfo*. Но не рассчитывайте разделаться с проектом за пять минут — интерфейсы библиотеки типов весьма сложны, и с ними придется повозиться.

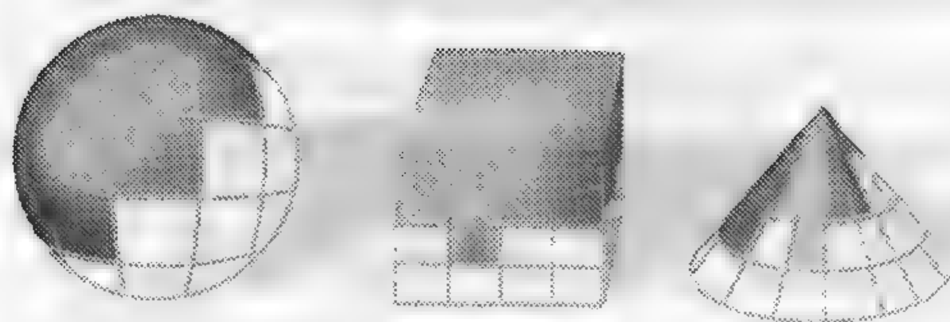
Повышение скорости связи контроллер-компонент

Microsoft признает ограничения интерфейса *IDispatch*. Он медлителен по природе, потому что все данные пропускаются через переменные типа *VARIANT* и зачастую преобразуются на обоих концах (на контроллере и в компоненте). Сейчас появилась новая его разновидность — *двойственный интерфейс* (dual interface), при работе с которым вы определяете собственный интерфейс, производный от *IDispatch*. В него включаются не только *Invoke* и *GetIDsOfNames*, но и новые функции. Если клиент достаточно «сообразителен», он обойдет неэффективный *Invoke* и вместо него вызовет специализированные функции. Двойственные интерфейсы могут поддерживать либо только стандартные типы данных Automation, либо произвольные типы[1]. [Обсуждение двойственных интерфейсов выходит за рамки этой книги — см. «Inside OLE 2» (Microsoft Press, 1995, Second Edition) Крейга Брокшмидта (Kraig Brockschmidt).]

Прямая поддержка двойственных интерфейсов в каркасе MFC среды Visual C++ .NET не предусмотрена, но пример ACDUAL, поставляемый с Visual C++, поможет получить первое представление о них.

---

[1] К методам и свойствам двойственного интерфейса можно обращаться либо через *IDispatch::Invoke*, либо через vtable. В последнем случае вызов выполняется быстрее. Однако типы данных, поддерживаемые двойственным интерфейсом, ограничены типами данных Automation. — *Прим. перев.*



ГЛАВА

24

Uniform Data Transfer: буфер обмена и операция OLE drag-and-drop

В технологии COM заложен мощный механизм обмена данными как внутри Windows-приложений, так и между приложениями. Он называется Uniform Data Transfer (унифицированная передача данных), или UDT. Как вы увидите, UDT обеспечивает все виды форматирования и хранения передаваемых данных, выходя далеко за рамки стандартной передачи данных через буфер обмена. Ключевой элемент механизма UDT в COM — интерфейс *IDataObject*.

MFC тоже поддерживает UDT, но не в такой степени, чтобы скрыть все, что происходит на уровне COM-интерфейсов. Одно из полезных применений унифицированной передачи данных — операция drag-and-drop («перетащить и отпустить»). Многие разработчики реализуют ее в приложениях как один из стандартных способов обмена информацией. Библиотека MFC тоже поддерживает операцию drag-and-drop, которая — вместе с передачей данных через буфер обмена — является основной темой этой главы.

Интерфейс *IDataObject*

Интерфейс *IDataObject* применяется при передаче данных не только через буфер обмена и операцией drag-and-drop, но и в составных документах, ActiveX-элементах и специализированных средствах OLE. В книге «Inside OLE» (Microsoft Press, 1995, Second Edition) Брокшмидт советует: «Рассматривайте объекты как крошечные пакеты данных». Так вот, интерфейс *IDataObject* помогает перемещать эти пакеты независимо от их содержимого.

Программируя на уровне Win32 API, вы написали бы класс C++ для поддержки *IDataObject*. Затем ваша программа создала бы *объекты данных* (data objects) этого класса, и вы обращались бы к ним через функции-члены *IDataObject*. В этой главе вы узнаете, как достичь того же результата, используя MFC-реализацию *IDataObject*. Но сначала выясним, чем OLE-буфер лучше обычного буфер обмена Windows.

Преимущества *IDataObject* в сравнении со стандартной поддержкой буфера обмена

MFC никогда особо не поддерживала буфер обмена Windows. Если вам уже приходилось работать с ним, вы наверняка вызывали Win32-функции *OpenClipboard*, *CloseClipboard*, *GetClipboardData* и *SetClipboardData*. Одна программа копирует данные в буфер обмена в определенном формате, а другая выбирает их оттуда по коду формата и вставляет в свой документ. Стандартные форматы буфера обмена включают участки глобальной памяти (определяемые по *HGLOBAL*) и GDI-объекты вроде растровых изображений и метафайлов (определяются их описателями). Глобальная память может содержать данные в нестандартных форматах и текст.

Интерфейс *IDataObject* дает то, чего не хватает буферу обмена Windows. Если коротко, то вы передаете или считываете из буфера обмена указатель на *IDataObject*, а не отдельные данные в каких-то форматах. Объект данных может содержать целый массив разных форматов, которые несут информацию об устройстве вывода, например характеристики принтера, и определять *представление данных* (data aspect). Стандартное представление — это сами данные, но есть и другие, например, значок или *микрокартинка* (thumbnail), отражающая хранимое изображение в уменьшенном виде.

Интерфейс *IDataObject* задает *носитель* объекта данных в конкретном формате. Обычная передача через буфер обмена опирается исключительно на использование глобальной памяти. С другой стороны, интерфейс *IDataObject* позволяет передавать имя дискового файла или указатель на *структурированное хранилище* (structured storage). Таким образом, при передаче очень большого объема данных, хранящихся в файле на диске, нет нужды тратить время на копирование его в оперативную память и обратно.

Между прочим, указатели *IDataObject* совместимы с программами, в которых применяются старые способы обмена даннми через буфер обмена. Коды форматов совпадают. Windows обеспечивает автоматическое преобразование информации в объекты данных, и наоборот. Разумеется, если программа, поддерживающая OLE, поместит в объект данных указатель на *IStorage* и передаст объект в буфер обмена, старые программы не смогут считать данные в этом формате.

Структуры *FORMATETC* и *STGMEDIUM*

Прежде чем приступить к изучению функций-членов *IDataObject*, нужно рассмотреть две важные структуры COM, используемые в качестве типов параметров: *FORMATETC* и *STGMEDIUM*.

FORMATETC

Эту структуру часто используют вместо формата буфера обмена для описания формата данных. В отличие от формата буфера она содержит информацию об устройстве вывода, представлении и носителе данных. Поля структуры *FORMATETC* описаны в таблице.

Тип	Имя	Описание
CLIPFORMAT	*cfFormat*	Структура, содержащая форматы буфера обмена: стандартные (например, *CF_TEXT* для текста или *CF_DIB* для сжатых изображений), нестандартные (например, RTF) и OLE (для создания связанных или внедренных объектов).
*DVTARGETDEVICE**	*ptd*	Структура, содержащая информацию об устройстве вывода данных, в том числе имя драйвера устройства (допустимое значение — *NULL*).
DWORD	*dwAspect*	Перечисляемая константа типа *DVASPECT* (*DVASPECT_CONTENT*, *DVASPECT_THUMBNAIL* и т. д.)
LONG	*lindex*	Обычно равно –1.
DWORD	*tymed*	Определяет носитель, используемый для передачи данных (*TYMED_HGLOBAL*, *TYMED_FILE*, *TYMED_ISTORAGE* и т. д.).

Конкретный объект данных содержит набор элементов *FORMATETC*, а *IDataObject* обеспечивает способ их перечисления. Вот макрос, полезный для заполнения этой структуры:

```
#define SETFORMATETC(fe, cf, asp, td, med, li) \
   ((fe).cfFormat=cf, \
    (fe).dwAspect=asp, \
    (fe).ptd=td, \
    (fe).tymed=med, \
    (fe).lindex=li)
```

Структура *STGMEDIUM*

Другая важная структура, используемая функциями-членами *IDataObject*, — *STGMEDIUM*, представляющая собой описатель глобальной памяти, используемой в передаче данных. Вот поля этой структуры.

Тип	Имя	Описание
DWORD	*tymed*	Определяет носитель; используется при маршалинге
HBITMAP	*hBitmap*	Описатель растрового изображения*
HMETAFILEPICT	*hMetaFilePict*	Описатель метафайла*
HENHMETAFILE	*hEnhMetaFile*	Описатель расширенного метафайла*
HGLOBAL	*hGlobal*	Описатель глобальной памяти*
LPOLESTR	*lpszFileName*	Имя файла на диске (двухбайтовые символы)*
*ISTREAM**	*pstm*	Указатель на интерфейс *IStream**
*ISTORAGE**	*pstg*	Указатель на интерфейс *IStorage**
*IUNKNOWN**	*pUnkForRelease*	Используется клиентами, чтобы вызвать *Release* для форматов с указателями на интерфейсы

* Это поле является частью *объединения* (union), в состав которого входят описатели, строки и указатели на интерфейсы, используемого процессом для доступа к передаваемым данным.

Как видите, структура *STGMEDIUM* определяет, *где* хранятся данные. Поле *tymed* сообщает, какой элемент объединения используется.

Функции-члены интерфейса *IDataObject*

У интерфейса *IDataObject* девять функций-членов. Все они детально описаны в книге Брокшмидта и в MFC Library Reference. Здесь упомянуты лишь те, что важны для понимания материала этой главы.

```
HRESULT EnumFormatEtc(DWORD dwDirection,
                      IEnumFORMATETC** ppEnum);
```

Имея указатель на *IDataObject* для объекта данных, можно вызвать *EnumFormatEtc* для перебора всех форматов, поддерживаемых этим объектом. Библиотека MFC изолирует вас от этого уродливого API. Как именно, вы узнаете при рассмотрении класса *COleDataObject*.

```
HRESULT GetData(FORMATETC* pFEIn, STGMEDIUM* pSTM);
```

GetData — самая важная функция интерфейса. Где-то «высоко-высоко в небесах» находится объект данных, а у вас есть указатель на его интерфейс *IDataObject*. В переменной *FORMATETC* вы задаете нужный формат и передаете пустую переменную *STGMEDIUM* для получения результатов. Если объект данных поддерживает заданный формат, *GetData* заполняет структуру *STGMEDIUM*. В противном случае возвращается код ошибки.

```
HRESULT QueryGetData(FORMATETC* pFE);
```

Эту функцию вызывают, когда не уверены, может ли объект данных предоставить данные в нужном формате. Код результата сообщает «да, могу» (*S_OK*) или «нет, не могу» (*S_FALSE*). Вызов этой функции эффективнее, чем вызов *GetData*.

```
HRESULT SetData(FORMATETC* pFEIn, STGMEDIUM* pSTM, BOOL fRelease);
```

Объекты данных редко поддерживают *SetData*. Обычно данные в них загружаются в серверном модуле, клиенты же выбирают данные вызовом *GetData*. Используя *SetData*, вы передавали бы данные в обратном направлении — все равно, что перекачивать воду из дома в водопровод.

Другие функции-члены *IDataObject*: консультативная связь

Этот интерфейс содержит другие важные функции, позволяющие реализовать *консультативные связи* (advisory connection). Программа, которой нужно получать уведомления об изменении данных объекта, передает ему указатель *IAdviseSink* вызовом *IDataObject::DAdvise*. Далее объект вызывает функции-члены *IAdviseSink*, реализуемые клиентской программой. Консультативные связи не нужны для операции drag-and-drop.

Поддержка механизма UDT в MFC

В библиотеке MFC сделано многое, чтобы облегчить программирование объектов данных. При изучении MFC-классов объектов данных вам станет понятен принцип, по которому построена поддержка COM в MFC. На стороне компонента биб-

лиотека обеспечивает базовый класс, реализующий один или несколько интерфейсов OLE. Функции-члены интерфейса вызывают виртуальные функции, которые вы переопределяете в производном классе, а на стороне клиента библиотека предоставляет класс-оболочку указателя на интерфейс. Вы вызываете простые функции-члены, использующие этот указатель для обращения к COM.

Здесь стоит внести ясность в терминологию. *Объект данных* (data object) — это настоящий объект C++, создаваемый вами, и именно в этом смысле данный термин использует в своей книге Брокшмидт. В документации по MFC объект данных — это то, что клиентское приложение «видит» через указатель на *IDataObject*. Объект, создаваемый программой компонента, называется там *источником данных* (data source).

Класс *COleDataSource*

Чтобы получить источник данных, вы создаете объект класса *COleDataSource*, который реализует интерфейс *IDataObject* (без поддержки уведомлений). Этот класс создает и управляет *набором* форматов данных, хранимым в кэше. Источник данных — это обычный COM-объект, поддерживающий счетчик ссылок. Обычно вы создаете и заполняете источник данных, после чего передаете его в буфер обмена или в функции, реализующие drag-and-drop, не беспокоясь о его дальнейшей судьбе. Если же вы решили не передавать источник данных, можете вызвать его деструктор, выполняющий очистку всех форматов. Далее приведены некоторые наиболее полезные функции-члены *COleDataSource*.

```
void CacheData(CLIPFORMAT cfFormat,
    STGMEDIUM* lpStgMedium,
    FORMATETC* lpFormatEtc = NULL);
```

Эта функция добавляет элемент к кэшу объекта данных для передачи данных. Параметр *lpStgMedium* указывает, где находятся данные, а *lpFormatEtc* описывает их. Например, если структура *STGMEDIUM* задает имя файла на диске, это имя и сохраняется в объекте данных. Если же *lpFormatEtc* не задан (т. е. его значение — NULL), функция заполняет структуру *FORMATETC* значениями по умолчанию. Но безопаснее, если вы создадите переменную *FORMATETC* с установленным значением поля *tymed*.

```
void CacheGlobalData(CLIPFORMAT cfFormat,
    HGLOBAL hGlobal, FORMATETC* lpFormatEtc = NULL);
```

Это специализированная версия *CacheData* для размещения данных в глобальной памяти (определяемой параметром *HGLOBAL*). Владельцем блока глобальных данных становится объект — источник данных, так что вам не надо освобождать блок после кэширования. Обычно параметр *lpFormatEtc* не применяют. Функция *CacheGlobalData* *не* создает копию переданных ей данных.

```
DROPEFFECT DoDragDrop(DWORD dwEffects =
    DROPEFFECT_COPY | DROPEFFECT_MOVE |
    DROPEFFECT_LINK, LPCRECT lpRectStartDrag = NULL,
    COleDropSource* pDropSource = NULL);
```

Эта функция вызывается для выполнения операции drag-and-drop; вы увидите ее в примере Ex24b.

```
void SetClipboard(void);
```

Эта функция (см. пример Ex24a), вызывает *OleSetClipboard* для вставки источника данных в буфер обмена Windows. Буфер обмена отвечает за удаление источника данных и тем самым за освобождение глобальной памяти, связанной с форматами в кэше. Когда вы создаете объект *COleDataSource* и вызываете *SetClipboard*, COM вызывает для объекта *AddRef*.

Класс *COleDataObject*

Этот класс (производный от *CCmdTarget*) является принимающей стороной в передаче данных; его открытая переменная-член *m_lpDataObject* указывает на *IDataObject*. Вы должны задать значение этой переменной прежде, чем использовать объект. Деструктор класса лишь вызывает *Release* для указателя на *IDataObject*. Далее приведено несколько полезных функций *COleDataObject*:

```
BOOL AttachClipboard(void);
```

Как указывает Брокшмидт, внутренняя обработка буфера обмена в OLE довольно сложна. Однако она покажется вам проще, если вызывать функции-члены *COleDataObject*. Сначала создается «пустой» объект *COleDataObject*, после чего вызывается функция *AttachClipboard*, которая обращается к глобальной функции *OleGetClipboard*. В результате переменная-член *m_lpDataObject* указывает на объект — источник данных (так это по крайней мере выглядит), и вы получаете доступ к его форматам.

Вызывая функцию-член *GetData*, чтобы получить данные, помните: владелец данных — буфер обмена, и вы не можете изменить его содержимое. Если форматом данных является указатель типа *HGLOBAL*, не пытайтесь освобождать эту память или запоминать *HGLOBAL* для дальнейшего использования. Если же вам нужен доступ к данным в глобальной памяти в течение длительного периода, подумайте об использовании *GetGlobalData*.

Если данные включены в буфер обмена программой, не поддерживающей COM, функция *AttachClipboard* все равно действует, так как COM автоматически создает объект данных с форматами, соответствующими обычным данным в буфере обмена Windows.

```
void BeginEnumFormats();
BOOL GetNextFormat(FORMATETC* lpFormatEtc);
```

Эти две функции позволяют просматривать в цикле форматы, которые содержит объект данных. Сначала вызывается *BeginEnumFormats*, после чего на каждой итерации вызывается *GetNextFormat*, пока не вернет *FALSE*.

```
BOOL GetData(CLIPFORMAT cfFormat,
    STGMEDIUM* lpStgMedium,
    FORMATETC* lpFormatEtc = NULL);
```

Эта функция вызывает *IDataObject::GetData* и более ничего. Если источник данных содержит запрашиваемый формат, функция возвращает *TRUE*. Обычно требуется указывать параметр *lpFormatEtc*.

```
HGLOBAL GetGlobalData(CLIPFORMAT cfFormat,
    FORMATETC* lpFormatEtc = NULL);
```

Вызывайте эту функцию, если запрашиваемый вами формат совместим с глобальной памятью. Функция копирует блок памяти в указанном формате и возвращает описатель *HGLOBAL*, который вы должны впоследствии освободить. Зачастую параметр *lpFormatEtc* можно не указывать.

```
BOOL IsDataAvailable(CLIPFORMAT cfFormat,
    FORMATETC* lpFormatEtc = NULL);
```

Эта функция проверяет, содержит ли объект данных указанный формат.

Передача объекта данных через буфер обмена

Теперь, познакомившись с классами *COleDataObject* и *COleDataSource*, вы можете запросто работать с буфером обмена. Но почему бы не передать данные через буфер обмена старым методом — с помощью *GetClipboardData* и *SetClipboardData*? Для наиболее распространенных форматов это допустимо, но, написав функции, работающие с объектами данных, вы сможете вызвать эти же функции и для поддержки операции drag-and-drop. На рис. 24-1 проиллюстрирована связь между буфером обмена и классами *COleDataSource* и *COleDataObject*.

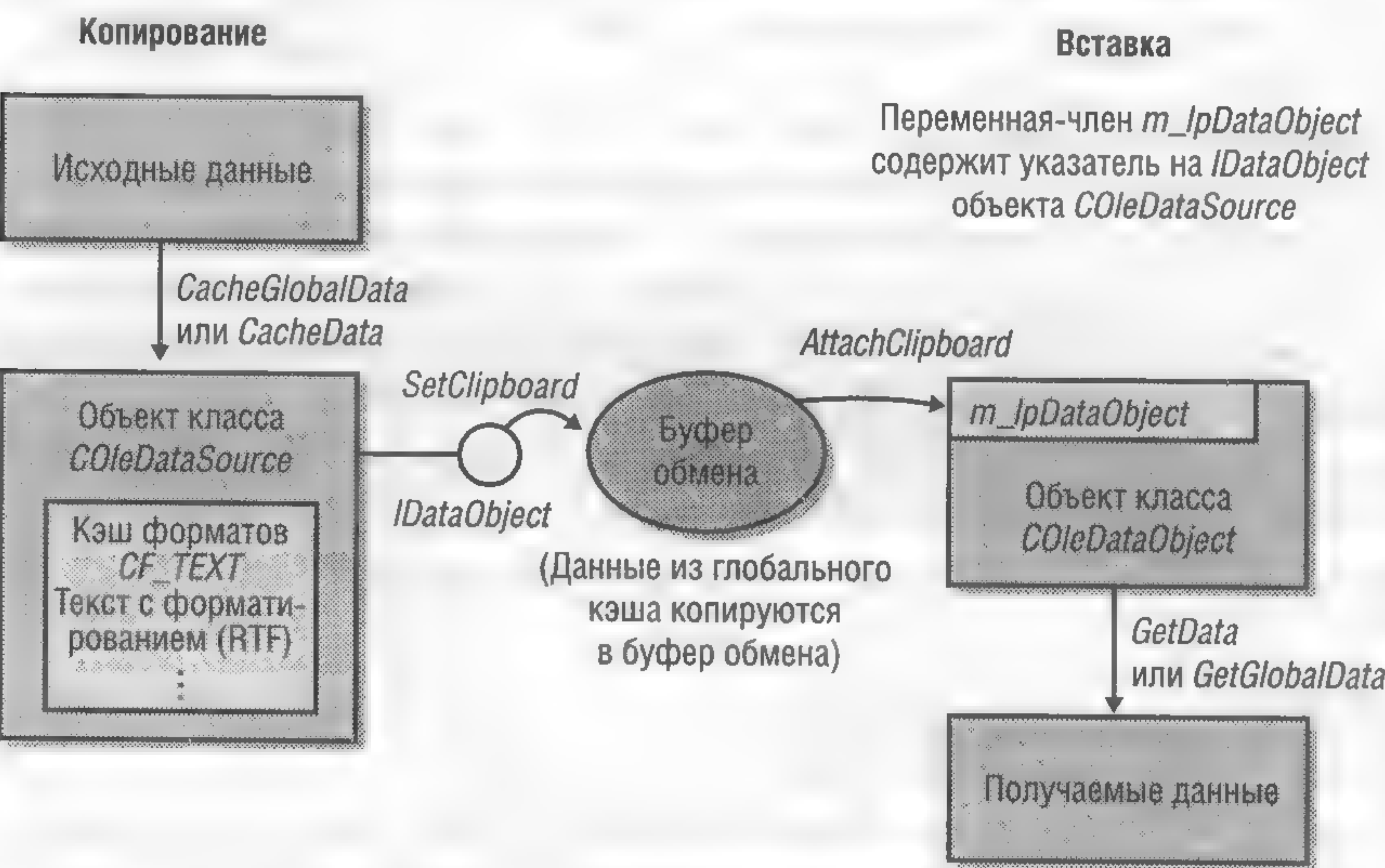


Рис. 24-1. *Обработка OLE-операций с буфером обмена в MFC*

Программа, копирующая данные, создает объект *COleDataSource*, кэш которого заполняется форматами. Потом вызывается *SetClipboard*, и форматы помещаются в буфер обмена. Программа, вставляющая данные, вызывает *AttachClipboard* для подключения указателя на *IDataObject* к объекту *COleDataObject*, а затем производит выборку отдельных форматов.

Допустим, в приложении в архитектуре «документ-вид» у документа есть переменная-член *m_strText* типа *CString*. Нам нужны функции — обработчики команд в классе «вид», которые копируют и вставляют данные из буфера обмена. Прежде чем писать эти функции, следует написать две вспомогательные. Первая, *SaveText*, создает объект — источник данных на основе содержимого *m_strText*. Она конструирует объект *COleDataSource*, затем копирует строку в глобальную память и, наконец, вызывает *CacheGlobalData*, чтобы сохранить в источнике данных описатель *HGLOBAL*. Вот код *SaveText*:

```
COleDataSource* CMyView::SaveText()
{
   CEx24fDoc* pDoc = GetDocument();
   if (!pDoc->m_strtext.IsEmpty()) {
      COleDataSource* pSource = new COleDataSource();
      int nTextSize = GetDocument()->m_strText.GetLength() + 1;
      HGLOBAL hText = ::GlobalAlloc(GMEM_SHARE, nTextSize);
      LPSTR pText = (LPSTR) ::GlobalLock(hText);
      ASSERT(pText);
      strncpy(pText, GetDocument()->m_strText,
            nTextSize - 1);
      ::GlobalUnlock(hText);
      pSource->CacheGlobalData(CF_TEXT, hText);
      return pSource;
   }
   return NULL;
}
```

Вторая вспомогательная функция, *DoPasteText*, заполняет *m_strText* содержимым объекта данных, переданного как параметр. Здесь мы используем *COleDataObject::GetData* вместо *GetGlobalData*, так как последняя копирует глобальный блок памяти. Это излишне, поскольку мы все равно скопируем текст в объект *CString*. Исходный блок памяти мы не освобождаем, потому что он принадлежит объекту данных. Вот код *DoPasteText*:

```
// Память является перемещаемой, поэтому надо использовать GlobalLock!
   SETFORMATETC(fmt, CF_TEXT, DVASPECT_CONTENT, NULL,
      TYMED_HGLOBAL, -1);
   VERIFY(pDataObject->GetData(CF_TEXT, &stg, &fmt));
   HGLOBAL hText = stg.hGlobal;
   GetDocument()->m_strText = (LPSTR) ::GlobalLock(hText);
   ::GlobalUnlock(hText);
   return TRUE;
}
```

А вот два обработчика команд:

```
void CMyView::OnEditCopy()
{
   COleDataSource* pSource = SaveText();
   if (pSource) {
      pSource->SetClipboard();
```

```
    }
}
void CMyView::OnEditPaste()
{
    COleDataObject dataObject;
    VERIFY(dataObject.AttachClipboard());
    DoPasteText(&dataObject);
    // Освобождаем dataObject
}
```

MFC-класс *CRectTracker*

Класс *CRectTracker* полезен для программ как применяющих, так и не применяющих OLE. Он дает пользователю возможность перемещать прямоугольный объект и изменять его размеры в окне представления. В нем есть две важных переменных-члена: *m_nStyle* (определяет рамку объекта, маркеры его масштабирования и прочие характеристики) и *m_rect* (содержит аппаратные координаты объекта). Рассмотрим наиболее важные функции-члены.

```
void Draw(CDC* pDC) const;
```

Рисует *прямоугольник* (tracker), ограничивающий объект, в том числе рамку и маркеры масштабирования, но не прорисовывает сам объект. Последнее — ваша задача.

```
BOOL Track(CWnd* pWnd, CPoint point,
      BOOL bAllowInvert = FALSE, CWnd* pWndClipTo = NULL);
```

Вызывается в обработчике сообщения *WM_LBUTTONDOWN*. Если курсор находится на рамке ограничивающего прямоугольника, пользователь может изменять размеры последнего, передвигая мышь при нажатой кнопке, а если курсор внутри прямоугольника — перемещать прямоугольник. Когда курсор оказывается за пределами прямоугольника, функция сразу же возвращает *FALSE*; в противном случае *Track* возвращает *TRUE*, но только после того, как пользователь отпустит кнопку мыши. Таким образом, поведение *Track* в чем-то схоже с поведением *CDialog::DoModal*. В нее заложена своя логика выборки сообщений.

```
int HitTest(CPoint point) const;
```

Вызывайте *HitTest*, если вам надо различать нажатия кнопки внутри и за пределами ограничивающего прямоугольника. Функция сразу возвращает управление и сообщает код, определяющий положение указателя мыши относительно прямоугольника.

```
BOOL SetCursor(CWnd* pWnd, UINT nHitTest) const;
```

Вызывайте ее в обработчике сообщения *WM_SETCURSOR* окна представления, чтобы изменять форму курсора при перемещении или масштабировании ограничивающего прямоугольника. Если возвращается *FALSE*, вызывайте функцию *OnSetCursor* из базового класса; а если *TRUE*, то возвращайте *TRUE* из своего обработчика.

Преобразование координат прямоугольника *CRectTracker*

CRectTracker::m_rect содержит аппаратные координаты. И вам не избежать преобразований, если вы используете прокрутку в окне представления, изменяете режим преобразования координат или смещаете начало координат окна. Общий способ решения этой задачи таков.

1. Определите в своем классе «вид» переменную-член типа *CRectTracker*, например *m_tracker*.
2. Определите в том же классе отдельную переменную-член для хранения логических координат с именем *m_rectTracker*.
3. В функции *OnDraw* класса «вид» запишите в *m_rect* новые аппаратные координаты и нарисуйте ограничивающий прямоугольник. Это позволит скорректировать изображение, если с момента предыдущего вызова *OnDraw* окно было прокручено. Вот пример кода:

```
m_tracker.m_rect = m_rectTracker;
pDC->LPtoDP(m_tracker.m_rect); // нужны аппаратные координаты
m_tracker.Draw(pDC);
```

4. В обработчике левой кнопки вызовите *Track*, сохраните обновленные логические координаты в *m_rectTracker* и вызовите *Invalidate*:

```
if (m_tracker.Track(this, point, FALSE, NULL)) {
    CClientDC dc(this);
    OnPrepareDC(&dc);
    m_rectTracker = m_tracker.m_rect;
    dc.DPtoLP(m_rectTracker);
    Invalidate();
}
```

Пример Ex24a: передача объекта данных через буфер обмена

В этом примере используется класс *CDib* из Ex06d. Вы сможете перемещать и масштабировать DIB-изображение с помощью ограничивающего прямоугольника, а также копировать DIB в буфер обмена и вставлять его оттуда в документ через объект данных COM. Кроме того, в примере реализованы функции для чтения и записи DIB в BMP-файлы.

Если вы создаете проект с самого начала, воспользуйтесь мастером MFC Application Wizard, сбросив все флажки, относящиеся к ActiveX и Automation, а потом добавьте в файл StdAfx.h строку:

```
#include <afxole.h>
```

а в начало тела функции *InitInstance* приложения — вызов:

```
AfxOleInit();
```

Чтобы подготовить Ex24a к работе, откройте и соберите проект \vcppnet\Ex24a\Ex24a.sln. Запустите программу и вставьте растровое изображение командой Paste From в меню Edit (рис. 24-2).

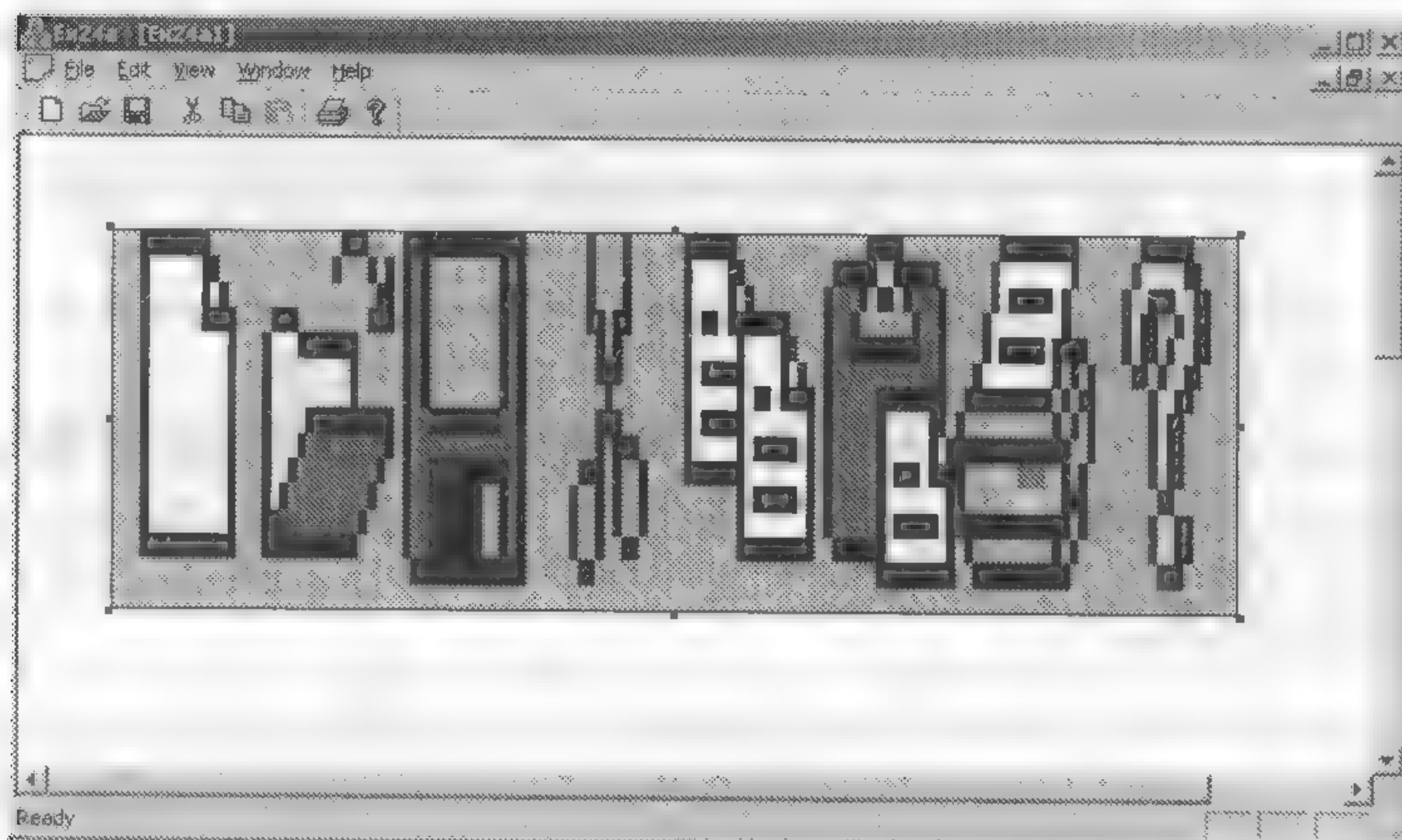


Рис. 24-2. *Программа Ex24a в действии*

Класс *CMainFrame*

Содержит обработчики *OnQueryNewPalette* и *OnPaletteChanged* сообщений *WM_QUERYNEWPALETTE* и *WM_PALETTECHANGED* соответственно. Эти обработчики отправляют пользовательское сообщение *WM_VIEWPALETTECHANGED* во все окна представления, затем вызывается *CDib::UsePalette* для реализации палитры. Значение *wParam* сообщает, должна ли палитра реализовываться в фоновом режиме или нет.

Класс *CEx24aDoc*

Весьма простой класс: содержит внедряемый *CDib*-объект *m_dib* и обработчик команды Clear All. Переопределенная функция-член *DeleteContents* вызывает функцию *CDib::Empty*.

Класс *CEx24aView*

Содержит обработчики команд, связанные с буфером обмена, код, отслеживающий поведение ограничивающего прямоугольника, а также код прорисовки DIB. Листинг класса приведен ниже; код, введенный вручную, выделен.

Ex24aView.h

```
// Ex24aView.h : interface of the CEx24aView class
//
#pragma once
#define WM_VIEWPALETTECHANGED WM_USER + 5
class CEx24aView : public CScrollView
{
    // Для отслеживания ограничивающего прямоугольника
    CRectTracker m_tracker;
```

```
    CRect m_rectTracker;        // логические координаты
    CSize m_sizeTotal;          // размер документа protected:
                                // создание только сериализацией
[illegible]
```

Ex24aView.cpp

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    [illegible]
[illegible]
[illegible]
[illegible] : m_sizeTotal(800, 1050), // 8x10,5 дюймов при печати
m_rectTracker(50, 50, 250, 250)
{
}
[illegible]
{
}
[illegible]
{
    [illegible]
    [illegible]
    [illegible]
}
[illegible]
[illegible]
{
    CDib& dib = GetDocument()->m_dib;
    m_tracker.m_rect = m_rectTracker;
    pDC->LPtoDP(m_tracker.m_rect); // нужны аппаратные координаты
    m_tracker.Draw(pDC);
```

```
    dib.Draw(pDC, m_rectTracker.TopLeft(), m_rectTracker.Size());
}
[illegible]
[illegible]
{
    pInfo->SetMaxPage(1);
    return DoPreparePrinting(pInfo);
}
[illegible]
{
    [illegible]
}
[illegible]
{
    [illegible]
}
[illegible]
[illegible]
[illegible]
{
    [illegible]
}
[illegible]
{
    [illegible]
}
[illegible]
{
    [illegible]
    [illegible]
}
[illegible]

[illegible]
BOOL CEx24aView::DoPasteDib(COleDataObject* pDataObject)
{
    // функции обновления командного пользовательского интерфейса
    // не должны допускать нас сюда, если CF_DIB недоступен
    if (!pDataObject->IsDataAvailable(CF_DIB)) {
        TRACE("CF_DIB format is unavailable\n");
        return FALSE;
    }
    CEx24aDoc* pDoc = GetDocument();
    // Это может быть ПЕРЕМЕЩАЕМАЯ память, поэтому мы должны использовать
    // GlobalLock! (hDib != lpDib). GetGlobalData копирует память, поэтому
    // мы можем использовать его, пока не удален CDib.
    HGLOBAL hDib = pDataObject->GetGlobalData(CF_DIB);
    ASSERT(hDib != NULL);
    LPVOID lpDib = ::GlobalLock(hDib);
```

см. след. стр.

```
    ASSERT(lpDib != NULL);
    pDoc->m_dib.AttachMemory(lpDib, TRUE, hDib);
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(NULL);
    return TRUE;
}
COleDataSource* CEx24aView::SaveDib()
{
    CDib& dib = GetDocument()->m_dib;
    if (dib.GetSizeImage() > 0) {
        COleDataSource* pSource = new COleDataSource();
        int nHeaderSize = dib.GetSizeHeader();
        int nImageSize = dib.GetSizeImage();
        HGLOBAL hHeader = ::GlobalAlloc(GMEM_SHARE,
            nHeaderSize + nImageSize);
        LPVOID pHeader = ::GlobalLock(hHeader);
        ASSERT(pHeader != NULL);
        LPVOID pImage = (LPBYTE) pHeader + nHeaderSize;
        memcpy(pHeader, dib.m_lpBMIH, nHeaderSize);
        memcpy(pImage, dib.m_lpImage, nImageSize);
        // Получатель должен освободить глобальную память
        ::GlobalUnlock(hHeader);
        pSource->CacheGlobalData(CF_DIB, hHeader);
        return pSource;
    }
    return NULL;
}
[illegible]
[illegible]
{
    COleDataSource* pSource = SaveDib();
    if (pSource) {
        pSource->SetClipboard(); // OLE deletes data source
    }
}
[illegible]
{
    // обслуживает Copy, Cut и Copy To
    CDib& dib = GetDocument()->m_dib;
    pCmdUI->Enable(dib.GetSizeImage() > 0L);
}
[illegible]
{
    OnEditCopy();
    GetDocument()->OnEditClearall();
}
[illegible]
{
    CEx24aDoc* pDoc = GetDocument();
    COleDataObject dataObject;
```

```
    VERIFY(dataObject.AttachClipboard());
    DoPasteDib(&dataObject);
    CClientDC dc(this);
    pDoc->m_dib.UsePalette(&dc);
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(NULL);
}
[illegible]
{
    COleDataObject dataObject;
    BOOL bAvail = dataObject.AttachClipboard() &&
        dataObject.IsDataAvailable(CF_DIB);
    pCmdUI->Enable(bAvail);
}
[illegible]
{
    CDib& dib = GetDocument()->m_dib;
    CFileDialog dlg(FALSE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) return;

    BeginWaitCursor();
    dib.CopyToMapFile(dlg.GetPathName());
    EndWaitCursor();
}
[illegible]
{
    CEx24aDoc* pDoc = GetDocument();
    CFileDialog dlg(TRUE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) return;
    if (pDoc->m_dib.AttachMapFile(dlg.GetPathName(), TRUE)) { // share
        CClientDC dc(this);
        pDoc->m_dib.SetSystemPalette(&dc);
        pDoc->m_dib.UsePalette(&dc);
        pDoc->SetModifiedFlag();
        pDoc->UpdateAllViews(NULL);
    }
}
[illegible]
{
    if (m_tracker.Track(this, point, FALSE, NULL)) {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        m_rectTracker = m_tracker.m_rect;
        dc.DPtoLP(m_rectTracker); // Обновляем логические координаты
        Invalidate();
    }
}
[illegible]
{
```

```
    if (m_tracker.SetCursor(pWnd, nHitTest)) {
        return TRUE;
    }
    else {
        return CScrollView::OnSetCursor(pWnd, nHitTest, message);
    }
}
[illegible]
{
    CScrollView::OnSetFocus(pOldWnd);
    AfxGetApp()->m_pMainWnd->SendMessage(WM_PALETTECHANGED,
        (UINT) GetSafeHwnd());
}
[illegible]
{
    // нестандартный MM_LOENGLISH; ось ординат y направлена вниз
    if (pDC->IsPrinting()) {
        int nHsize = pDC->GetDeviceCaps(HORZSIZE) * 1000 / 254;
        int nVsize = pDC->GetDeviceCaps(VERTSIZE) * 1000 / 254;
        pDC->SetMapMode(MM_ANISOTROPIC);
        pDC->SetWindowExt(nHsize, nVsize);
        pDC->SetViewportExt(pDC->GetDeviceCaps(HORZRES),
                            pDC->GetDeviceCaps(VERTRES));
    }
    else {
        CScrollView::OnPrepareDC(pDC, pInfo);
    }
}
[illegible]
{
    SetScrollSizes(MM_TEXT, m_sizeTotal);
    m_tracker.m_nStyle = CRectTracker::solidLine |
        CRectTracker::resizeOutside;
    CScrollView::OnInitialUpdate();
}
[illegible]
{
    TRACE("CEx24aView::OnViewPaletteChanged, HWND = %x, code = %d\n",
          GetSafeHwnd(), wParam);
    CClientDC dc(this);
    GetDocument()->m_dib.UsePalette(&dc, wParam);
    Invalidate();
    return 0;
}
```

В классе «вид» происходит несколько интересных вещей. Во вспомогательной функции *DoPasteDib* мы можем вызвать *GetGlobalData*, потому что можно подсоединить возвращаемую переменную типа *HGLOBAL* к объекту *CDib* в документе. Если бы мы вызвали *GetData*, нам пришлось бы самим копировать блок памяти. Обработчики команд Paste From и Copy To опираются на поддержку проецируемых в

память файлов, заложенную в класс *CDib*. Функция *OnPrepareDC* создает особый режим преобразования координат на принтере, совпадающий с *MM_LOENGLISH* за исключением того, что ось *y* направлена вниз (т. е. положительные значения координаты по оси откладываются вниз). Одна точка на экране соответствует 0,01 дюйма на принтере.

Поддержка операции drag-and-drop в MFC

Самый веский аргумент в пользу кода объекта данных — возможность поддержки операции drag-and-drop. OLE предоставляет для этого интерфейсы *IDropSource* и *IDropTarget*, а также библиотечный код, управляющий процессом перемещения объектов. Библиотека MFC поддерживает drag-and-drop на уровне класса «вид», чем мы и воспользуемся. Учтите, что в процессе выполнения drag-and-drop данные передаются напрямую и независимо от буфера обмена. Если пользователь отменяет операцию, никакой информации о перемещаемом объекте не сохраняется.

С точки зрения пользователя, передача данных операцией drag-and-drop между приложениями, окнами в одном приложении и в пределах конкретного окна ничем не отличается. Когда начинается эта операция, форма курсора должна изменяться на стрелку с прямоугольником. Если пользователь удерживает клавишу Ctrl, знак плюс (+) в изображении указателя информирует, что объект копируется, а не перемещается.

MFC поддерживает также операцию drag-and-drop для элементов составных документов. Это следующий, более высокий уровень поддержки OLE в MFC, который в этой главе не рассматривается. Подробнее см. пример OCLIENT в разделе Visual C++ Samples в MSDN Library.

Что происходит в источнике

Начиная операцию drag-and-drop для объекта данных, программа-источник вызывает *COleDataSource::DoDragDrop*. Эта функция создает объект MFC-класса *COleDropSource*, реализующего интерфейс *IDropSource*. *DoDragDrop* — из числа тех функций, что возвращают управление не сразу. Возврат из нее происходит, только когда пользователь «отпускает» объект или отменяет операцию либо когда проходит заданное количество миллисекунд.

Если вы программируете drag-and-drop так, чтобы эта операция работала с объектом *CRectTracker*, то *DoDragDrop* следует вызывать, только если пользователь щелкает мышью *внутри* ограничивающего прямоугольника, но не на рамке. Нужную информацию даст функция *CRectTracker::HitTest*. Перед вызовом *DoDragDrop* следует установить флаг, который подскажет, не «отпущен» ли объект там же, откуда началось его перемещение.

Что происходит в приемнике

Чтобы задействовать для drag-and-drop поддержку из класса «вид» библиотеки MFC, добавьте в свой производный класс «вид» переменную-член класса *COleDropTarget*. Класс *COleDropTarget* реализует интерфейс *IDropTarget* и хранит указатель на *IDropSource*, обеспечивающий обратную связь с объектом *COleDropSource*. В функ-

ции *OnInitialUpdate* вашего класса «вид» вызовите функцию-член *Register* для внедряемого объекта *COleDropTarget*.

Создав *COleDropTarget* для своего класса «вид», переопределите четыре виртуальные функции *CView*, которые MFC вызывает в процессе выполнения drag-and-drop. Вот что должны делать эти функции, если вы применяете класс *CRectTracker*:

Функции	Описание
OnDragEnter	Корректирует прямоугольник фокуса и вызывает *OnDragOver*.
OnDragOver	Перемещает пунктирный прямоугольник фокуса и определяет момент «отпускания» объекта (задает форму курсора).
OnDragLeave	Отменяет операцию и возвращает исходные размеры и координаты прямоугольника.
OnDrop	Корректирует прямоугольник фокуса и вызывает вспомогательную функцию *DoPaste*, чтобы извлечь форматы из объекта данных.

Последовательность действий при drag-and-drop

Рис. 24-3 иллюстрирует обработку операции drag-and-drop в MFC.

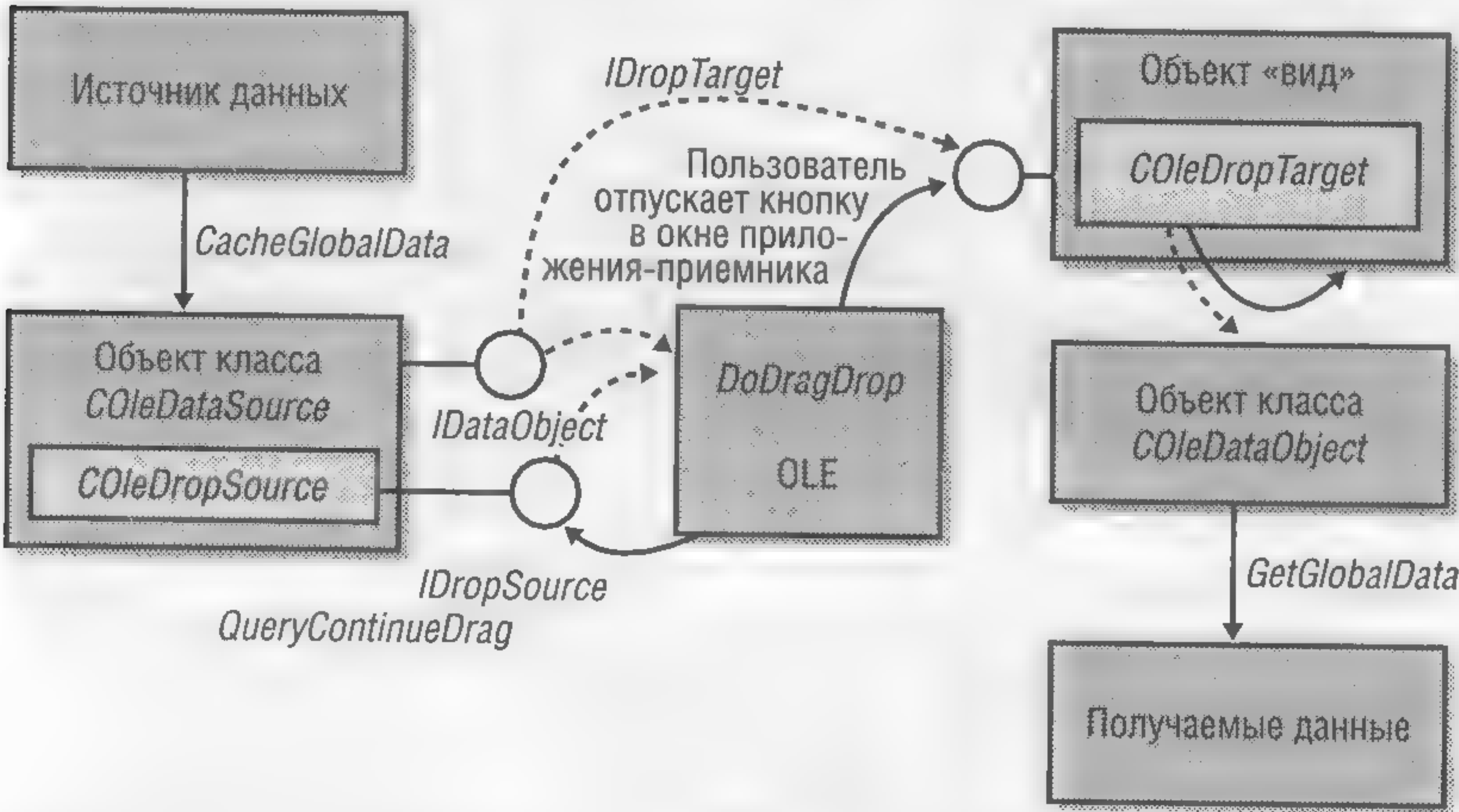


Рис. 24-3. *Обработка операции drag-and-drop в MFC*

Вот что происходит в процессе выполнения drag-and-drop.

1. Пользователь нажимает левую кнопку в окне представления источника.
2. Обработчик кнопки вызывает *CRectTracker::HitTest* и выясняет, расположен ли курсор внутри ограничивающего прямоугольника.
3. Обработчик сохраняет форматы в объекте *COleDataSource*.
4. Обработчик вызывает *COleDataSource::DoDragDrop* для источника данных.
5. Пользователь перемещает курсор в окно представления приемника.
6. OLE вызывает *IDropTarget::OnDragEnter* и *OnDragOver* для объекта *COleDropTarget*, в результате чего вызываются соответствующие виртуальные функции класса «вид» приемника. Функции *OnDragOver* передается указатель на *COleDataObject* для объекта-источника, который получатель использует для проверки наличия поддерживаемых им форматов.

7. *OnDragOver* возвращает код эффекта «отпускания», используемый OLE для установки формы курсора.
8. OLE вызывает *IDropSource::QueryContinueDrag* на стороне источника, чтобы определить, продолжать ли операцию перемещения. MFC-класс *COleDataSource* реагирует соответственно.
9. Пользователь отпускает кнопку мыши, чтобы «оставить» объект в окне представления приемника.
10. OLE вызывает *IDropTarget::OnDrop*, а та — *OnDrop* класса «вид» приемника. Так как *OnDrop* передается указатель на *COleDataObject*, она может выбрать из этого объекта данные в нужном формате.
11. Когда *OnDrop* программы-приемника возвращает управление, *DoDragDrop* в программе-источнике тоже может вернуть управление.

Пример Ex24b: OLE drag-and-drop

Эта программа принимает эстафету у Ex24a. В нее добавлена поддержка drag-and-drop с использованием уже имеющихся вспомогательных функций *SaveDib* и *DoPasteDib*. Код для работы с буфером обмена тот же. Этот пример можно адаптировать для других приложений, требующих применения drag-and-drop для объектов данных.

Для подготовки Ex24b откройте и соберите решение \vcppnet\Ex24b\Ex24b.sln. Запустите приложение и попробуйте что-нибудь переместить мышью между его дочерними окнами и между копиями этой программы.

Класс *CEx24bDoc*

Этот класс почти такой же, как *CEx24aDoc*, кроме новой переменной-члена *m_bDragHere*, используемой как флаг. Если его значение равно *TRUE*, идет операция drag-and-drop для данного документа. Флажок задан для документа, а не для окна представления, так как у одного документа может быть несколько представлений. Перемещать DIB из одного окна представления в другое бессмысленно, если оба они ссылаются на *m_dib* из одного документа.

Класс *CEx24bView*

В этом классе три дополнительных переменных-члена и конструктор, их инициализирующий:

```
CRect m_rectTrackerEnter; // исходные логические координаты
COleDropTarget m_dropTarget;
CSize m_dragOffset; // аппаратные координаты

CEx24bView::CEx24bView() : m_sizeTotal(800, 1050), // 8x10,5 дюймов при печати
    m_rectTracker(50, 50, 250, 250),
    m_dragOffset(0, 0),
    m_rectTrackerEnter(50, 50, 250, 250)
{
}
```

В функции *OnInitialUpdate* нужна дополнительная строка для регистрации получателя:

```
m_dropTarget.Register(this);
```

А теперь посмотрим на переопределенные виртуальные функции для drag-and-drop. Заметьте, что *OnDrop* заменяет DIB, только если флажок *m_bDragHere* в документе равен *TRUE*, поэтому, когда пользователь «отпустит» DIB в том же окне (или в другом окне, подсоединенном к тому же документу), ничего не произойдет.

```
DROPEFFECT CEx24bView::OnDragEnter(COleDataObject* pDataObject,
                               DWORD dwKeyState, CPoint point)
{
   TRACE("Entering CEx24bView::OnDragEnter, point = (%d, %d)\n",
         point.x, point.y);
   m_rectTrackerEnter = m_rectTracker; // Сохранить начальные координаты на случай,
                                       // если объект не "отпустят" на новом месте
   CClientDC dc(this);
   OnPrepareDC(&dc);
   dc.DrawFocusRect(m_rectTracker); // будет стерт в OnDragOver
   return OnDragOver(pDataObject, dwKeyState, point);
}
void CEx24bView::OnDragLeave()
{
   TRACE("Entering CEx24bView::OnDragLeave\n");
   CClientDC dc(this);
   OnPrepareDC(&dc);
   dc.DrawFocusRect(m_rectTracker);
   m_rectTracker = m_rectTrackerEnter; // Восстановление начальных координат
}
DROPEFFECT CEx24bView::OnDragOver(COleDataObject* pDataObject, DWORD
                          dwKeyState, CPoint point)
{
   if (!pDataObject->IsDataAvailable(CF_DIB)) {
      return DROPEFFECT_NONE;
   }
   MoveTrackRect(point);
   if((dwKeyState & MK_CONTROL) == MK_CONTROL) {
      return DROPEFFECT_COPY;
   }
   // Проверка на принудительное перемещение
   if ((dwKeyState & MK_ALT) == MK_ALT) {
      return DROPEFFECT_MOVE;
   }
   // По умолчанию рекомендуется предполагать перемещение
   return DROPEFFECT_MOVE;
}
BOOL CEx24bView::OnDrop(COleDataObject* pDataObject,
                 DROPEFFECT dropEffect, CPoint point)
{
   TRACE("Entering CEx24bView::OnDrop - dropEffect = %d\n", dropEffect);
   BOOL bRet;
```

```
    CEx24bDoc* pDoc = GetDocument();
    MoveTrackRect(point);
    if(pDoc->m_bDragHere) {
        pDoc->m_bDragHere = FALSE;
        bRet = TRUE;
    }
    else {
        bRet = DoPasteDib(pDataObject);
    }
    return bRet;
}
```

Обработчик сообщения *WM_LBUTTONDOWN* требует существенной модификации. Он должен вызывать *DoDragDrop*, если курсор находится в прямоугольнике, и *Track* — если на его рамке. Вот новый вариант кода:

```
void CEx24bView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CEx24bDoc* pDoc = GetDocument();
    if(m_tracker.HitTest(point) == CRectTracker::hitMiddle) {
        COleDataSource* pSource = SaveDib();
        if(pSource) {
            // DoDragDrop возвращает управление только по окончании перемещения
            CClientDC dc(this);
            OnPrepareDC(&dc);
            CPoint topleft = m_rectTracker.TopLeft();
            dc.LPtoDP(&topleft);
            // 'point' здесь не то же самое, что параметр point в OnDragEnter,
            // поэтому мы используем его для вычисления смещения
            m_dragOffset = point - topleft;  // аппаратные координаты
            pDoc->m_bDragHere = TRUE;
            DROPEFFECT dropEffect = pSource->DoDragDrop(
                DROPEFFECT_MOVE|DROPEFFECT_COPY, CRect(0, 0, 0, 0));
            TRACE("after DoDragDrop - dropEffect = %ld\n", dropEffect);
            if (dropEffect == DROPEFFECT_MOVE && pDoc->m_bDragHere) {
                pDoc->OnEditClearall();
            }
            pDoc->m_bDragHere = FALSE;
            delete pSource;
        }
    }
    else {
        if(m_tracker.Track(this, point, FALSE, NULL)) {
            CClientDC dc(this);
            OnPrepareDC(&dc);
            // надо как-то предотвратить выход за границы
            m_rectTracker = m_tracker.m_rect;
            dc.DPtoLP(m_rectTracker); // Обновляем логические координаты
        }
    }
    Invalidate();
}
```

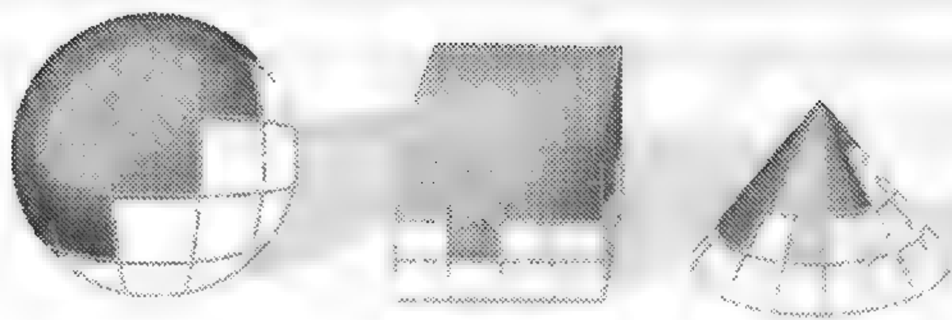
И, наконец, новая вспомогательная функция *MoveTrackRect*, показанная ниже, перемещает прямоугольник фокуса при каждом вызове *OnDragOver*. В примере Ex24a эту работу выполняла *CRectTracker::Track*.

```
void CEx24bView::MoveTrackRect(CPoint point)
{
    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.DrawFocusRect(m_rectTracker);
    dc.LPtoDP(m_rectTracker);
    CSize sizeTrack = m_rectTracker.Size();
    CPoint newTopleft = point - m_dragOffset; // по-прежнему аппаратные координаты
    m_rectTracker = CRect(newTopleft, sizeTrack);
    m_tracker.m_rect = m_rectTracker;
    dc.DPtoLP(m_rectTracker);
    dc.DrawFocusRect(m_rectTracker);
}
```

Мы тестировали Ex24b с пакетом Microsoft Office XP, опробовав передачу данных как через drag-and-drop, так и через буфер обмена. Формат *CF_DIB* здесь отсутствует. Так что, если вы хотите вставлять картинки из Excel, в Ex24b надо ввести поддержку метафайлов.

ГЛАВА

25



Основы ATL

В этой главе мы рассмотрим второй (если первым считать MFC) каркас приложений, в составе Microsoft Visual C++ .NET — Active Template Library (ATL). Для начала вспомним COM, затем рассмотрим альтернативный способ написания объекта *CSpaceship* из главы 22, чтобы показать разные методы написания COM-класса. (Это будет важно при рассмотрении методов композиции классов ATL.) Далее мы изучим ATL, сосредоточившись сначала на шаблонах и обычных smart-указателях C++ и их применении в разработке COM объектов. Потом мы обсудим программирование на ATL со стороны клиента и некоторые из smart-указателей ATL. Наконец, мы придем к программированию серверов на ATL, заново реализовав с ее помощью пример космического корабля из главы 22, что даст нам представление об архитектуре ATL.

Снова COM

Краеугольный камень COM — интерфейсы. Как вы узнали из главы 22, ни COM, ни какая-либо поддержка периода выполнения не нужны для программирования при помощи интерфейсов. Все, что вам нужно, — это дисциплина.

Вернемся к примеру космического корабля из главы 22. Мы начали с одного класса *CSpaceship*, реализующего несколько функций. Опытные разработчики на C++ обычно начинают писать класс примерно так:

```
class CSpaceship {
   void Fly();
   int& GetPosition();
};
```

Однако при разработке на основе интерфейсов процедура немного отличается. Вместо написания класса напрямую интерфейс сначала обговаривается, а по-

том реализуется. Ранее функции *Fly* и *GetPosition* переводились в абстрактный базовый класс *IMotion*.

```
struct IMotion {
    virtual void Fly() = 0;
    virtual int& GetPosition() = 0;
};
```

Затем класс *CSpaceship* объявлялся производным от интерфейса *IMotion*:

```
class CSpaceship : IMotion {
    void Fly();
    int& GetPosition();
};
```

Обратите внимание: интерфейс *движения* (motion) отделен от своей реализации. При разработке интерфейсов вы можете изменять их, стремясь сделать полными, но в то же время не слишком раздутыми. Но как только интерфейс опубликован (т. е. другие разработчики начали его использовать), он замораживается и никогда не изменяется.

Такое небольшое различие между программированием на основе классов и на основе интерфейсов создает некоторые проблемы. Однако это один из ключевых моментов для понимания COM. Собрав вместе функции *Fly* и *GetPosition*, вы разработали двоичную структуру. Если есть предварительно определенный интерфейс, то общение с классом через интерфейс позволяет клиенту получить потенциально не зависящий от языка способ общения с классом.

Группировка функций в интерфейс сама по себе очень полезна. Допустим, вы хотите описать что-нибудь отличающееся от космического корабля, скажем, самолет. Ясно, что у него также должны быть функции *Fly* и *GetPosition*. Программирование на основе интерфейсов предоставляет более широкую форму полиморфизма — полиморфизм на уровне интерфейсов, а не на уровне одной функции.

Разработка на основе интерфейсов базируется на отделении интерфейса от реализации. COM ориентирована на программирование интерфейсов и заставляет разделять интерфейс и реализацию. Единственный способ общения клиента с объектом согласно COM — интерфейс. Однако сведения функций вместе в интерфейс недостаточно. Нужен еще один ингредиент — механизм для выяснения функциональности во время выполнения.

Базовый интерфейс *IUnknown*

Основное правило, отличающее COM от обычного программирования на основе интерфейсов, заключается в том, что названия первых трех функций совпадают во всех COM-интерфейсах. Базовый интерфейс COM — *IUnknown* — выглядит так:

```
struct IUnknown {
    virtual HRESULT QueryInterface(REFIID riid, void** ppv) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

Все COM-интерфейсы наследуют именно ему (что делает *QueryInterface*, *AddRef* и *Release* первыми тремя функциями всех COM-интерфейсов). Для преобразования *IMotion* в интерфейс COM его нужно преобразовать в интерфейс, производный от *IUnknown*:

```
struct IMotion : IUnknown {
   void Fly();
   int& GetPosition();
};
```

Примечание Если вы хотите, чтобы интерфейс работал за пределами процесса, сделайте у каждой функции возвращаемое значение типа HRESULT. Подробнее об ATL с атрибутами мы поговорим позже.

AddRef и *Release* требуют некоторого внимания, так как являются частью *IUnknown*. Они позволяют объекту контролировать свое время жизни. Как правило, клиенты трактуют указатели на интерфейсы как ресурсы: клиенты запрашивают интерфейсы, используют их и освобождают, когда те становятся ненужными. Объекты узнают о новых ссылках на себя через *AddRef*, а об освобождении — через *Release*. Объекты часто используют эту информацию для управления своей «жизнью». Например, многие объекты самоуничтожаются, когда их счетчик ссылок обнуляется.

Вот как код клиента мог бы использовать космический корабль:

```
void UseSpaceship() {
   IMotion* pMotion = NULL;

   pMotion = GetASpaceship(); // Это функция гипотетического Spaceship API.
               // Она скорее всего является точкой входа в некоторую DLL,
               // возвращает IMotion* и выполняет неявный вызов AddRef.
   If(pMotion) {
      pMotion->Fly();
      int i = pMotion->GetPosition();
      pMotion->Release(); // закончили с этим экземпляром CSpaceship
   }
}
```

Последняя (и самая важная) функция *IUnknown* — это его первая функция *QueryInterface*. Она является механизмом выяснения функциональности во время выполнения. Если вы получили указатель на интерфейс и не хотите использовать именно его, то вы можете при помощи указателя запросить у объекта другой интерфейс. Данный механизм, а также тот факт, что интерфейсы остаются неизменными с момента публикации, позволяют ПО на основе COM безопасно эволюционировать во времени. В результате вы можете добавлять функциональность в ПО без переделки старых версий клиентов, использующих это ПО. Кроме того, у клиентов есть широко известные средства получения этой функциональности, если они о ней знают. Например, вы добавили функциональность в *CSpaceship*, реализовав новый интерфейс *IVisual*. Такой интерфейс имеет смысл, потому что у вас есть объект, находящийся в трехмерном пространстве и двигающийся к или

от наблюдателя. Кроме того, у вас может быть невидимый объект (черная дыра, например). *IVisual* может выглядеть так:

```
struct IVisual : IUnknown {
   virtual void Display() = 0;
};
```

Клиент может использовать *IVisual* следующим образом:

```
void UseSpaceship() {
   IMotion* pMotion = NULL;

   pMotion = GetASpaceship(); // Неявный вызов AddRef
   if(pMotion) {
      pMotion->Fly();
      int i = pMotion->GetPosition();

      IVisual* pVisual = NULL;
      pMotion->QueryInterface(IID_IVisual, (void**) &pVisual);
      // Неявный вызов AddRef внутри QueryInterface

      if(pVisual) {
         pVisual->Display(); // теперь видимый
         pVisual->Release(); // закончить с этим интерфейсом
      }
   }
   pMotion->Release(); // закончить с этим экземпляром IMotion
}
```

В этом коде указатели на интерфейсы используются очень осторожно: они применяются, только если интерфейс получен корректно, и освобождаются, когда интерфейсы более не нужны. Это обычное низкоуровневое COM-программирование: вы запрашиваете указатель на интерфейс, используете указатель на интерфейс, освобождаете его, когда он более не нужен.

Написание COM-кода

Как вы видели, написание кода COM-клиента не сильно отличается от написания обычного кода C++. Однако классы C++, с которыми имеет дело клиент, — это базовые абстрактные классы. Вместо вызова *operator new*, как это было в C++, вы создаете COM-объекты и запрашиваете COM-интерфейсы путем явного вызова нескольких API-функций; а вместо удаления объекта вы просто следуете правилу COM о равенстве числа вызовов *AddRef* и *Release*.

Что требуется для создания работающего COM-класса? Вы видели в главе 22, как сделать такой класс при помощи MFC. Здесь приводится другой пример реализации *CSpaceship* как COM-класса, в котором используется множественное наследование. Класс C++ наследует несколько интерфейсов и реализует все их функции (включая *IUnknown*, естественно).

```
struct CSpaceship : IMotion, IDisplay {
   ULONG m_cRef;
```

```
    int m_nPosition;

    CSpaceship() : m_cRef(0),
                   m_nPosition(0) {
    }

    HRESULT QueryInterface(REFIID riid, void** ppv);
    ULONG AddRef() {
        return InterlockedIncrement(&m_cRef);
    }
    ULONG Release() {
        ULONG cRef = InterlockedIncrement(&m_cRef);
        if(cRef == 0){
            delete this;
            return 0;
        }
        else
            return m_cRef;
    }

    // Функции IMotion:
    void Fly() {
        // Здесь располагается весь необходимый для полета код
    }
    int GetPosition() {
        return m_nPosition;
    }

    // Функции IVisual:
    void Display() {
        // Показать
    }
};
```

COM-классы на основе множественного наследования

Если вы привыкли к обычному коду C++, то предыдущий код может показаться вам слегка странным. Это не совсем обычная форма множественного наследования, называемая *наследование интерфейса* (interface inheritance). Большинство разработчиков на C++ пользуется *наследованием реализации* (implementation inheritance), при котором производный класс наследует от базового все, в том числе его реализацию. Наследование интерфейса просто означает, что производный класс наследует интерфейсы базового класса. Код, приведенный выше, добавляет в класс *CSpaceship* две переменные-члены — указатели на каждою невидимую, но подразумеваемую *vtable.*

В случае множественного наследования для реализации интерфейсов все они совместно использует реализацию *IUnknown* в *CSpaceship*. Это иллюстрирует другое известное только посвященным, но важное понятие — речь идет о *единстве в смысле COM* (COM identity). Основная идея единства в COM состоит в том, что

IUnknown в COM — это то же, что *void** в C++. *IUnknown* — единственный интерфейс, который гарантированно есть в любом объекте, и указатель на него всегда можно получить. Клиент может вызвать *QueryInterface* через интерфейс *IMotion* объекта *CSpaceship* для получения интерфейса *IVisible*. И наоборот, клиент может вызвать *QueryInterface* через интерфейс *IVisible* объекта *CSpaceship* для получения интерфейса *IMotion*. Наконец, клиент может вызвать *QueryInterface* через интерфейс *IUnknown* для получения *IMotion* или *IVisible* или через *IMotion* или *IVisible* для получения *IUnknown*. О единстве в смысле COM см. книги Дона Бокса (Don Box) «Essential COM» (Addison-Wesley, 1997) и Дейла Роджерсона (Dale Rogerson) «Inside COM» (Microsoft Press, 1997) (Роджерсон Д. Основы COM. 2-е издание. М.: Русская Редакция, 2000).

Часто можно видеть COM-классы в виде прямоугольников с кружочками интерфейсов, реализованных в этих классах. Пример такой диаграммы приведен в главе 22 в разделе «Интерфейс *IUnknown* и функция-член *QueryInterface*».

Множественное наследование при реализации *CSpaceship* автоматически удовлетворяет правилам единства в COM. Заметьте: все вызовы *QueryInterface*, *AddRef* и *Release* передаются в одни и те же функции-члены класса C++ независимо от интерфейса, через который они вызваны.

Вот более или менее вся сущность COM. Вы, как разработчик, создаете полезные сервисы и предоставляете их клиенту через COM-интерфейсы. На самом низком уровне это означает наличие нескольких таблиц функций, удовлетворяющих правилам COM. К настоящему времени вы видели два способа реализации этого. (В главе 22 показано, как это сделать при помощи вложенных классов и MFC. В данной главе мы только что написали COM-класс при помощи множественного наследования.) Однако в COM есть еще несколько важных вопросов, помимо программирования интерфейсов и написания классов для их реализации.

Инфраструктура COM

Хотя концепция программирования на основе интерфейсов вам известна, есть довольно много деталей, которые нужно реализовать, чтобы ваш класс стал частью системы. Эти детали часто затемняют фундаментальную красоту COM.

Для начала COM-классам нужно место для жизни, поэтому вы должны поместить их в EXE или DLL. Кроме того, каждому COM-классу нужен соответствующий объект класса, часто называемый *фабрикой класса* (class factory). Способ обращения к объекту класса COM-сервера зависит от размещения самого класса COM (в DLL или EXE). Необходимо также учесть время жизни сервера. Сервер должен оставаться в памяти, пока он нужен, и удаляться, как только в нем больше нет необходимости. Для этого серверы содержат глобальные счетчики блокировок, соответствующие количеству объектов, имеющих указатели на интерфейсы. Наконец, корректно работающие серверы добавляют в реестр Windows параметры, облегчающие запуск клиентов.

Как вы помните, MFC заботится о большинстве деталей программирования на основе COM (см. главу 22). Так, в *CCmdTarget* есть реализация *IUnknown*. MFC даже создает классы C++ и макросы, реализующие объекты класса (например, *COleObjectFactory*, *COleTemplateServer*, *DECLARE_OLE_CREATE* и *IMPLEMENT_OLE_CREATE*),

которые помещают в реестр большинство нужных записей. В MFC есть самая простая в использовании и быстрая версия *IDispatch*, поэтому все, что вам нужно, — это объект *CCmdTarget* и среда Visual Studio .NET (мастера Add Property Wizard и Add Method Wizard). Если вам требуется OLE drag-and-drop, MFC предоставляет стандартную реализацию этой операции. Наконец, MFC, бесспорно, остается простейшей технологией для написания быстрых и мощных элементов управления ActiveX. (Их можно писать и на Microsoft Visual Basic, но этот язык не обеспечивает достаточной гибкости.)

Это были достоинства. Теперь о недостатках. Во-первых, для использования всех этих возможностей нужно знать MFC на все сто. Во-вторых, решившись на применение MFC, надо представлять себе цену этого решения. MFC огромна. Она должна быть такой, поскольку является каркасом приложений на C++ с большим количеством возможностей.

Как вы могли видеть из ранее рассмотренных примеров, реализация COM-классов и предоставление клиентам доступа к ним требует написания большого объема кода — кода, который не изменяется от реализации одного класса к реализации другого. Например, реализация *IUnknown* обычно одна и та же для любого COM-класса — основное различие между ними в интерфейсах, предоставляемых каждым классом.

Прежде чем «нырнуть» в глубины ATL, бросим беглый взгляд на общую картину, частью которой являются COM и ATL.

ActiveX, OLE и COM

COM — это всего лишь «арматура» для нескольких высокоуровневых технологий, таких как элементы управления ActiveX и OLE drag-and-drop. Можно реализовать высокоуровневые возможности OLE-документов и элементов управления самостоятельно, однако разумнее предоставить эту работу какому-нибудь каркасу приложений, в первую очередь MFC.

Примечание Подробнее о реализации высокоуровневых функций в «сыром» C++ см. книгу Крейга Брокшмидта (Kraig Brockschmidt) «Inside OLE» (Microsoft Press, Second Edition, 1995).

ActiveX, MFC и COM

Хотя COM-«арматура» интересна сама по себе (например, просто изумительно наблюдать, как работает распределенная COM), именно высокоуровневые возможности создают продаваемые приложения. MFC — это огромный каркас, предназначенный для создания целых Windows-приложений. В MFC вы найдете «тонны» полезных классов, механизм управления-представления данных (архитектура «документ-вид»), поддержку drag and drop, Automation и ActiveX-элементов. Вам скорее всего не захочется с нуля разрабатывать приложение OLE drag and drop — лучше использовать MFC. Однако если необходимо создать небольшой или средних размеров сервис на основе COM, то вы вряд ли захотите тащить в него весь «багаж», который MFC применяет для поддержки высокоуровневых возможностей.

Для создания COM-компонентов можно использовать обычный код на C++, но тогда большая часть времени уйдет на написание стереотипного кода (например, для интерфейса *IUnknown* и объектов класса). Применение MFC для создания приложений на основе COM — наименее болезненный способ добавить крупные элементы в приложение, но на MFC тяжело писать простые COM-классы. ATL позиционируется между чистым C++ и MFC как способ реализовать ПО на основе COM без написания стереотипного кода или изучения всей архитектуры MFC. ATL представляет собой набор шаблонов C++ и других средств, облегчающих написание классов COM.

Путеводитель по ATL

Если вы посмотрите на исходный код библиотеки ATL, то увидите, что вся она содержится в нескольких файлах C++, большинство из которых находится в подкаталоге ATLMFC\Include каталога с Microsoft Visual Studio .NET. Далее приведено описание некоторых файлов ATL и того, что находится в них.

AtlBase.h

Этот файл хранит:

- объявления функций ATL;
- определения структур и макросов;
- описания smart-указателей, управляющих указателями на COM-интерфейсы;
- классы поддержки синхронизации потоков;
- определения классов: *CComBSTR*, *CComVariant*, поддержки потоков и разделения потоков.

AtlCom.h

В этом файле находятся:

- шаблоны классов для поддержки фабрик классов (объектов класса);
- реализации интерфейса *IUnknown*;
- поддержка для *обособленных* (tear-off) интерфейсов;
- поддержка получения информации о типе;
- реализация *IDispatch*;
- шаблоны классов-перечислителей;
- поддержка точек соединения.

AtlConv.cpp и AtlConv.h

Эти файлы обеспечивают поддержку преобразования Unicode-строк в ANSI и обратно.

AtlCtl.cpp и AtlCtl.h

В этих двух файлах хранятся:

- исходный код ATL для поддержки *IDispatch* со стороны клиента и поддержки генерации событий;

- ComControlBase;
- поддержка механизма внедрения объектов;
- поддержка страниц свойств.

AtlIFace.idl и AtlIFace.h

AtlIFace.idl (и генерируемый из него AtlIFace.h) содержит специальный ATL-интерфейс *IRegistrar*.

AtlImpl.cpp

Реализует некоторые классы, например *CComBSTR*, объявленный в AtlBase.h.

AtlWin.cpp и AtlWin.h

Эти файлы предоставляют поддержку для работы с окнами и пользовательским интерфейсом, включая:

- механизм карт сообщений;
- оконный класс;
- диалоговые окна.

StatReg.cpp и StatReg.h

ATL предоставляет COM-компонент *Registrar*, который выполняет регистрацию в реестре. Код реализации находится в StatReg.h и StatReg.cpp.

А теперь начнем нашу «экскурсию» по ATL с изучения поддержки разработки COM-клиентов.

Программирование клиента с помощью ATL

Есть две стороны ATL: клиентская и серверная. Большая часть поддержки программирования находится на стороне сервера, так как ATL содержит весь код, необходимый для реализации элементов управления ActiveX. Однако поддержка для клиентов, предоставляемая ATL, тоже весьма интересна и полезна. Давайте рассмотрим клиентскую сторону ATL. Поскольку шаблоны C++ являются краеугольным камнем ATL, мы сначала поговорим о их.

Шаблоны C++

Несмотря на пугающий синтаксис, концепция шаблонов очень проста. Иногда их называют *дружественными компилятору макросами* (compiler-approved macros), что достаточно точно описывает их сущность. Вспомните: когда препроцессор встречает макрос, он подставляет его содержимое в код C++. Недостаток макросов в том, что они зачастую содержат ошибки и в них полностью отсутствует проверка типов. Если вы передадите в макрос некорректный параметр, компилятор не «заметит» этого, но ваша программа может очень легко «сломаться». Шаблоны похожи на макросы с проверкой типов. Когда компилятор встречает шаблон, он подставляет его содержимое подобно макросу, однако выполняет при этом проверку типов, тем самым избавляя пользователя от многих проблем.

Применение шаблонов для повторного использования кода отличается от всего, что вы делали в обычной разработке на C++. Компоненты, написанные на основе шаблонов, повторно используют код путем подстановки параметров шаблона, а не наследуя функциональность базовых классов. Весь стереотипный код из шаблонов полностью вставляется в проект.

Типовой пример применения шаблона — динамический массив. Допустим, вам нужен массив для хранения целых чисел, но вам не хочется объявлять массив фиксированного размера, так как его длину придется увеличивать по мере надобности. Вы создаете массив в виде класса C++. Затем ваш коллега говорит, что ему нужен такой же массив, но для чисел с плавающей запятой. Вместо того чтобы создать тот же самый код, но с другим типом данных, вы можете применить шаблон C++.

Вот пример решения описанной задачи — динамический массив, реализованный как шаблон:

```
template <class T> class DynArray {
public:
   DynArray();
   ~DynArray();              // очистка и операции по управлению памятью
   int Add(T Element);       // добавление элемента и выполнение операций
                             // по управлению памятью
   void Remove(int nIndex)   // удаление элемента и выполнение операций
                             // по управлению памятью
   T GetAt(nIndex) const;
   int GetSize();
private:
   T* TArray;
   int m_nArraysize;
};

void UseDynArray() {
   DynArray<int> intArray;
   DynArray<float> floatArray;

   intArray.Add(4);
   floatArray.Add(5.0);

   intArray.Remove(0);
   floatArray.Remove(0);

   int x = intArray.GetAt(0);
   float f = floatArray.GetAt(0);
}
```

Ясно, что создание шаблонов полезно для реализации стереотипного кода COM, и ATL использует их именно так. Приведенный пример — лишь один из многих способов применения шаблонов. Однако шаблоны позволяют не только указать информацию о типе структур данных — они удобны для инкапсуляции алгоритмов. Вы это увидите, когда ближе познакомитесь с ATL. Изучение ATL начнем со smart-указателей.

Smart-указатели

Одно из типовых применений шаблонов — smart-указатели («интеллектуальные» указатели). В литературе по «традиционному» C++ обычные указатели называют «*бессловесными*» (dumb). Звучит не слишком ласково, но обычные указатели практически ничего не делают — лишь указывают на что-то. Деталями, такими как инициализация указателя, клиент занимается сам.

В качестве примера давайте смоделируем с помощью C++ два класса, имитирующих поведение программистов, один из которых пишет на Visual Basic, а второй — на C++. Начнем с создания классов *CVBDeveloper* и *CCPPDEveloper*.

```
class CVBDeveloper {
public:
   CVBDeveloper() {
   }
   ~CVBDeveloper() {
      AfxMessageBox("Я использую Visual Basic .NET, поэтому ухожу домой пораньше.");
   }
   virtual void DoTheWork() {
      AfxMessageBox("Пишу формы.");
   }
};

class CCPPDeveloper {
public:
   CCPPDeveloper() {
   }
   ~CCPPDeveloper() {
      AfxMessageBox("Остался на работе и решаю проблемы с указателями.");
   }
   virtual void DoTheWork() {
      AfxMessageBox("Разбираюсь в коде на C++.");
   }
};
```

У обоих разработчиков есть функции для достижения оптимальной производительности. Теперь представьте себе некий клиент примерно такого вида:

```
//UseDevelopers.cpp

void UseDevelopers() {
   CVBDeveloper* pVBDeveloper;
   ⋮
   // Указатель на VB Developer нужно где-то инициализировать.
   // Но что, если вы забыли проинициализировать
   // и позднее случится нечто вроде:
   if(pVBDeveloper) {
      // Готовьтесь к худшему.
      // Так как pVBDeveloper НЕ РАВЕН NULL, он указывает
      // на какие-то случайные данные.
      c->DoTheWork();
   }
}
```

В данном случае клиент забыл присвоить *NULL* указателю *pVBDeveloper*. (Да-да, этого никогда не бывает в реальной жизни!) Так как *pVBDeveloper* содержит ненулевое значение (просто значение, которое оказалось в стеке в этот момент), то проверка указателя будет успешной, хотя она должна быть неудачной. Клиент «радостно» продолжит выполнение, веря, что все хорошо. Конечно, программа «рухнет», так как вызов уйдет в «пустоту». (Кто знает, на что указывает *pVBDeveloper*, — вряд ли на что-то, похожее на VB-разработчика.) Естественно, вам понравится механизм, который всегда инициализирует указатели. Вот где оказываются полезными smart-указатели.

Теперь представьте себе другой сценарий. Вы хотите добавить в классы, описывающие разработчики, немного кода, который выполнял бы общие для всех разработчиков операции. Например, вы желаете, чтобы все разработчики сначала создали проект, а потом начали кодировать. Посмотрите на вышеприведенные примеры разработчиков на VB и на C++. При вызове *DoTheWork* разработчик начнет кодирование без соответствующего проекта, и, вероятно, оставит несчастный клиент в беде. Вам хотелось бы добавить в классы разработчиков универсальную *функцию-перехватчик* (hook), чтобы удостовериться в завершении проектирования до начала кодирования. В C++ решение этих проблем называется *smart-указателем*. Что же это такое?

Наполнение указателей «интеллектом»

Smart-указатель — это созданный на C++ класс-оболочка обычного указателя. Создав класс-оболочку (точнее, шаблон), вы автоматизируете определенные операции вместо того, чтобы вынуждать клиент содержать стереотипный код. Один из примеров такой операции — инициализация указателя, дабы не было случайно «рухнувших» программ из-за «приблудных» указателей. Другой пример — выполнение заданного кода перед вызовом функции с помощью указателя.

Давайте создадим smart-указатель для ранее описанной модели разработчиков. Вот класс-шаблон *SmartDeveloper*:

```
template<class T>
class SmartDeveloper {
   T* m_pDeveloper;

public:
   SmartDeveloper(T* pDeveloper) {
      ASSERT(pDeveloper != NULL);
      m_pDeveloper = pDeveloper;
   }
   ~SmartDeveloper() {
      AfxMessageBox("Я умный, поэтому получу зарплату.");
   }
   SmartDeveloper &
      operator=(const SmartDeveloper& rDeveloper) {
      return *this;
   }
    T* operator->() const {
      AfxMessageBox("К разыменованию указателя. /
                    Проверяю, все ли в порядке.");
```

```
        return m_pDeveloper;
    }
};
```

Шаблон *SmartDeveloper* инкапсулирует указатель — *любой* указатель. Он может предоставить типовую функциональность независимо от типа указателя, ассоциированного с ним. Относитесь к шаблонам, как к дружественным для компилятора макросам: объявлениям классов (или функций), код которых применим к данным любого типа.

Мы хотим, чтобы smart-указатель был применим к любым разработчикам, в том числе пишущим на VB, Visual C++, Java и Delphi. Это достигается указанием в начале определения класса оператора *template <class T>*. В шаблоне объявлен также указатель (*m_pDeveloper*) на тип разработчика, для которого определен класс. Конструктор получает указатель этого типа как параметр и присваивает его *m_pDeveloper*. Заметьте: конструктор генерирует предупреждение, если клиент передает параметр, равный NULL.

В дополнение к инкапсуляции указателя *SmartDeveloper* реализует несколько операторов. Наиболее важным из них является «->» (оператор выбора функции-или переменной-члена) — «рабочая лошадка» любого класса smart-указателя. Именно переопределение оператора выбора превращает обычный класс в smart-указатель. Обычное использование этого оператора для «бессловесного» указателя C++ говорит компилятору, что нужно вызвать метод класса (или структуры), на который ссылается указатель. Переопределив оператор, вы позволяете перехватить вызов и выполнить стереотипный код при всяком вызове метода. В классе *SmartDeveloper* «умный» разработчик проверяет рабочее место перед началом работы. (Это немного искусственный пример. В реальной жизни можно, например, вставить отладочный код.)

Добавление «->» в класс заставляет класс вести себя, как встроенный указатель C++. Чтобы быть похожим на указатели языка C++ во всем остальном, класс smart-указателя должен реализовать другие стандартные операторы — разыменования и присваивания.

Применение smart-указателей

Применение smart-указателей не отличается от применения обычных встроенных указателей языка C++. Начнем с клиента, использующего готовые (неизмененные) классы разработчиков:

```
void UseDevelopers() {
    CVBDeveloper VBDeveloper;
    CCPPDeveloper CPPDeveloper;

    VBDeveloper.DoTheWork();
    CPPDeveloper.DoTheWork();
}
```

Никаких сюрпризов — выполнение этого кода заставит разработчиков прийти и выполнить работу. Однако вам нужны «умные» разработчики, т. е. такие, которые убедятся в наличии проекта прежде, чем начать кодирование. Вот код, который инкапсулирует классы разработчиков в оболочку-класс smart-указателя:

```
void UseSmartDevelopers {
    CVBDeveloper VBDeveloper;
    CCPPDeveloper CPPDeveloper;

    SmartDeveloper<CVBDeveloper> smartVBDeveloper(&VBDeveloper);
    SmartDeveloper<CCPPDeveloper> smartCPPDeveloper(&CPPDeveloper);

    smartVBDeveloper->DoTheWork();
    smartCPPDeveloper->DoTheWork();
}
```

Вместо того чтобы обращаться к старым разработчикам (как в предыдущем примере), клиент заставляет «интеллектуалов» автоматически подготовить проект перед началом кодирования.

Smart-указатели и COM

Хотя последний пример служил всего лишь для придания большей динамики рассказу, smart-указатели полезны и в реальном мире. Одно из таких применений — облегчение программирования COM-клиентов.

Smart-указатели часто используют при реализации подсчета ссылок. Перенос подсчета ссылок на стороне клиента в smart-указатели имеет смысл, поскольку это одна из базовых операций COM.

Вы уже знаете, что COM-объекты предоставляют интерфейсы. Для клиентов на C++ интерфейсы — это просто чисто абстрактные базовые классы, поэтому интерфейсы часто трактуются как более-менее обычные классы C++. Однако, как вы помните, COM-объекты слегка отличаются от обычных объектов C++ тем, что существуют на двоичном уровне. Это означает, что они создаются и уничтожаются с использованием независимых от языка средств. COM-объекты создаются через вызовы функций API. Многие COM-объекты подсчитывают ссылки для определения момента, когда они могут самоуничтожиться. После создания объекта клиент может обращаться к нему множеством способов через его интерфейсы. Кроме того, с одним объектом может взаимодействовать несколько клиентов. В этих ситуациях COM-объект должен оставаться в памяти, пока на него есть ссылки. Для определения момента самоуничтожения служит подсчет ссылок.

Для поддержки схемы подсчета ссылок COM определяет несколько правил управления COM-интерфейсами со стороны клиента. Первое правило: копирование интерфейса должно увеличивать счетчик ссылок объекта на единицу[1]. Второе: клиент должен освободить указатель на интерфейс по окончании работы с ним. Подсчет ссылок — одна из труднейших для правильной реализации особенностей COM, особенно со стороны клиента, и поэтому является первым кандидатом на применение smart-указателей.

Например, конструктор такого указателя мог бы получать в качестве аргумента указатель на реальный интерфейс и сохранять его во внутреннем указателе. Деструктор мог бы вызывать функцию *Release* для интерфейса, чтобы автомати-

---

[1] Клиент обязан предполагать, что у каждого интерфейса свой счетчик ссылок, и должен вызвать *AddRef* именно для копируемого интерфейса. — *Прим. перев.*

чески освободить интерфейс, когда smart-указатель удаляется или выходит из области видимости.

Кроме того, smart-указатель может поддерживать копирование интерфейсов COM. Например, представьте, что вы создали COM-объект и храните указатель на интерфейс. Пусть вам нужно скопировать указатель на интерфейс (чтобы возвратить его как выходной параметр). На уровне обычной COM нужно выполнить несколько действий. Сначала надо освободить старый указатель. Затем присвоить старому указателю новый. Наконец, вызвать *AddRef* для новой копии указателя. Эти действия нужно выполнять независимо от используемого интерфейса, что делает описанный процесс идеальным кандидатом для шаблонного кода. Чтобы реализовать копирование в классе smart-указателя, нужно всего лишь переопределить оператор присваивания, после чего клиент может просто присвоить старому указателю новый. Smart-указатель выполнит все операции с указателем на интерфейс, облегчив бремя клиента.

Smart-указатели в ATL

Большая часть поддержки в ATL COM-программирования клиентов находится в двух классах smart-указателей: *CComPtr* и *CComQIPtr*. *CComPtr* — это базовый smart-указатель, инкапсулирующий указатель на COM-интерфейс. *CComQIPtr* чуть более интеллектуален благодаря сопоставлению GUID (используемого как идентификатор интерфейса) со smart-указателем. Большая часть функциональности *CComPtr* вынесена в класс *CComPtrBase*. Рассмотрим сначала *CComPtrBase*.

Класс *CComPtrBase*

Этот класс лежит в основе классов smart-указателей, работающих с основанными на COM функциями управления памятью. Вот листинг *CComPtrBase*.

```
template <class T>
class CComPtrBase
{
protected:
    CComPtrBase() throw()
    {
        p = NULL;
    }
    CComPtrBase(int nNull) throw()
    {
        ATLASSERT(nNull == 0);
        (void)nNull;
        p = NULL;
    }
    CComPtrBase(T* lp) throw()
    {
        p = lp;
        if (p != NULL)
            p->AddRef();
    }
public:
```

```
typedef T _PtrClass;
~CComPtrBase() throw()
{
   if (p)
      p->Release();
}
operator T*() const throw()
{
   return p;
}
T& operator*() const throw()
{
   ATLASSERT(p!=NULL);
   return *p;
}
// Оператор контроля (assert), примененный к operator&,
// обычно свидетельствует об ошибке. Однако если это то,
// что нужно, явно принимаем адрес переменной-члена p.
T** operator&() throw()
{
   ATLASSERT(p==NULL);
   return &p;
}
_NoAddRefReleaseOnCComPtr<T>* operator->() const throw()
{
   ATLASSERT(p!=NULL);
   return (_NoAddRefReleaseOnCComPtr<T>*)p;
}
bool operator!() const throw()
{
   return (p == NULL);
}
bool operator<(T* pT) const throw()
{
   return p < pT;
}
bool operator==(T* pT) const throw()
{
   return p == pT;
}
// Освобождаем интерфейс и устанавливаем переменную в NULL
void Release() throw()
{
   T* pTemp = p;
   if (pTemp)
   {
      p = NULL;
      pTemp->Release();
   }
}
```

```
// Проверяем два объекта на предмет эквивалентности
bool IsEqualObject(IUnknown* pOther) throw()
{
   if (p == pOther)
      return true;

   if (p == NULL || pOther == NULL)
      return false; // Один равен NULL, а второй - нет

   CComPtr<IUnknown> punk1;
   CComPtr<IUnknown> punk2;
   p->QueryInterface(__uuidof(IUnknown), (void**)&punk1);
   pOther->QueryInterface(__uuidof(IUnknown), (void**)&punk2);
   return punk1 == punk2;
}
// Поключение к существующему интерфейсу (не вызывает AddRef)
void Attach(T* p2) throw()
{
   if (p)
      p->Release();
   p = p2;
}
// Отключение от интерфейса (не вызывает Release)
T* Detach() throw()
{
   T* pt = p;
   p = NULL;
   return pt;
}
HRESULT CopyTo(T** ppT) throw()
{
   ATLASSERT(ppT != NULL);
   if (ppT == NULL)
      return E_POINTER;
   *ppT = p;
   if (p)
      p->AddRef();
   return S_OK;
}
HRESULT SetSite(IUnknown* punkParent) throw()
{
   return AtlSetChildSite(p, punkParent);
}
HRESULT Advise(IUnknown* pUnk, const IID& iid, LPDWORD pdw) throw()
{
   return AtlAdvise(p, pUnk, iid, pdw);
}
HRESULT CoCreateInstance(REFCLSID rclsid,
                         LPUNKNOWN pUnkOuter = NULL,
                         DWORD dwClsContext = CLSCTX_ALL) throw()
```

```
    {
        ATLASSERT(p == NULL);
        return ::CoCreateInstance(rclsid, pUnkOuter, dwClsContext,
                                  __uuidof(T), (void**)&p);
    }
    HRESULT CoCreateInstance(LPCOLESTR szProgID,
                             LPUNKNOWN pUnkOuter = NULL,
                             DWORD dwClsContext = CLSCTX_ALL) throw()
    {
        CLSID clsid;
        HRESULT hr = CLSIDFromProgID(szProgID, &clsid);
        ATLASSERT(p == NULL);
        if (SUCCEEDED(hr))
            hr = ::CoCreateInstance(clsid, pUnkOuter, dwClsContext,
                                    __uuidof(T), (void**)&p);
        return hr;
    }
    template <class Q>
    HRESULT QueryInterface(Q** pp) const throw()
    {
        ATLASSERT(pp != NULL);
        return p->QueryInterface(__uuidof(Q), (void**)pp);
    }
    T* p;
};
```

CComPtrBase — довольно простой smart-указатель. Обратите внимание на переменную-член *p* типа *T* — типа, определенного параметром шаблона. Конструктор выполняет *AddRef* для указателя, а деструктор освобождает указатель. Пока ничего из ряда вон выходящего. В *CComPtrBase* тоже есть все операторы, необходимые для инкапсуляции COM-интерфейса. Особого внимания заслуживает оператор присваивания, в котором выполняется переприсвоение исходного указателя путем вызова функции *AtlComPtrAssign*:

```
ATLINLINE ATLAPI_(IUnknown*) AtlComPtrAssign(IUnknown** pp,
                                             IUnknown* lp)
{
    if (lp != NULL)
        lp->AddRef();
    if (*pp)
        (*pp)->Release();
    *pp = lp;
    return lp;
}
```

Функция выполняет «слепое» присваивание указателя, вызывая *AddRef* для нового указателя перед вызовом *Release* для старого. В дальнейшем мы познакомимся с версией этой функции, которая вызывает *QueryInterface*.

Главное преимущество *CComPtrBase* в том, что он облегчает управление счетчиком ссылок на указатель. Следующий класс ниже в иерархии — *CComPtr*. Именно он применяется в «реальных» приложениях.

Класс *CComPtr*

Поскольку *CComPtr* происходит от *CComPtrBase*, он наследует все возможности последнего по управлению указателями на интерфейс. *CComPtr* облегчает управление операциями *AddRef* и *Release* и структурой кода. Вот небольшой пример, демонстрирующий полезность *CComPtr*. Пусть для выполнения определенных действий вам нужны три указателя на интерфейсы:

```
void GetLottaPointers(LPUNKNOWN pUnk){
   HRESULT hr;
   LPPERSIST pPersist;
   LPDISPATCH pDispatch;
   LPDATAOBJECT pDataObject;
   hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
   if(SUCCEEDED(hr)) {
      hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *)
                       &pDispatch);
      if(SUCCEEDED(hr)) {
         hr = pUnk->QueryInterface(IID_IDataObject,
                          (LPVOID *) &pDataObject);
         if(SUCCEEDED(hr)) {
            DoIt(pPersist, pDispatch, pDataObject);
            pDataObject->Release();
         }
         pDispatch->Release();
      }
      pPersist->Release();
   }
}
```

Вы можете использовать (рискуя нарваться на насмешливые комментарии своих коллег) очень непопулярный оператор *goto* для создания более понятного кода:

```
void GetLottaPointers(LPUNKNOWN pUnk){
   HRESULT hr;
   LPPERSIST pPersist;
   LPDISPATCH pDispatch;
   LPDATAOBJECT pDataObject;

   hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
   if(FAILED(hr)) goto cleanup;

   hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &pDispatch);
   if(FAILED(hr)) goto cleanup;

   hr = pUnk->QueryInterface(IID_IDataObject, (LPVOID *) &pDataObject);
   if(FAILED(hr)) goto cleanup;

   DoIt(pPersist, pDispatch, pDataObject);

cleanup:
   if (pDataObject) pDataObject->Release();
```

```
    if (pDispatch) pDispatch->Release();
    if (pPersist) pPersist->Release();
}
```

Такое решение может показаться не слишком элегантным. Применение *CComPtr* делает код красивее и понятнее:

```
void GetLottaPointers(LPUNKNOWN pUnk){
    HRESULT hr;
    CComPtr<IUnknown> persist;
    CComPtr<IUnknown> dispatch;
    CComPtr<IUnknown> dataobject;

    hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&persist);
    if(FAILED(hr)) return;

    hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &dispatch);
    if(FAILED(hr)) return;

    hr = pUnk->QueryInterface(IID_IDataObject, (LPVOID *) &dataobject);
    if(FAILED(hr)) return;

    DoIt(pPersist, pDispatch, pDataObject);

    // Деструкторы вызовут Release...
}
```

Сейчас у вас может возникнуть вопрос: почему *CComPtr* не инкапсулирует *QueryInterface*? В конце концов *QueryInterface* — это основное место подсчета ссылок. Однако для добавления поддержки *QueryInterface* в smart-указатель нужно как-то ассоциировать с ним GUID. Поскольку *CComPtr* существовал еще в первой версии ATL, Microsoft не стала изменять его код, а вместо этого добавила улучшенную версию — класс *CComQIPtr*.

Класс *CComQIPtr*

Вот определение *CComQIPtr*:

```
template <class T, const IID* piid = &__uuidof(T)>
class CComQIPtr : public CComPtr<T>
{
public:
    CComQIPtr() throw()
    {
    }
    CComQIPtr(T* lp) throw() :
        CComPtr<T>(lp)
    {
    }
    CComQIPtr(const CComQIPtr<T,piid>& lp) throw() :
        CComPtr<T>(lp.p)
    {
```

```
    }
    CComQIPtr(IUnknown* lp) throw()
    {
        if (lp != NULL)
            lp->QueryInterface(*piid, (void **)&p);
    }
    T* operator=(T* lp) throw()
    {
        return static_cast<T*>(AtlComPtrAssign((IUnknown**)&p, lp));
    }
    T* operator=(const CComQIPtr<T,piid>& lp) throw()
    {
        return static_cast<T*>(AtlComPtrAssign((IUnknown**)&p, lp.p));
    }
    T* operator=(IUnknown* lp) throw()
    {
        return static_cast<T*>(AtlComQIPtrAssign((IUnknown**)&p,
                        lp, *piid));
    }
};

// Специализация класса
template<>
class CComQIPtr<IUnknown, &IID_IUnknown> : public CComPtr<IUnknown>
{
public:
    CComQIPtr() throw()
    {
    }
    CComQIPtr(IUnknown* lp) throw()
    {
        // Запрос интерфейсов
        if (lp != NULL)
            lp->QueryInterface(__uuidof(IUnknown), (void **)&p);
    }
    CComQIPtr(const CComQIPtr<IUnknown,&IID_IUnknown>& lp) throw() :
        CComPtr<IUnknown>(lp.p)
    {
    }
    IUnknown* operator=(IUnknown* lp) throw()
    {
        // Запрос интерфейсов
        return AtlComQIPtrAssign((IUnknown**)&p, lp,
                        __uuidof(IUnknown));
    }
    IUnknown* operator=(const CComQIPtr<IUnknown,&IID_IUnknown>& lp)
        throw()
    {
        return AtlComPtrAssign((IUnknown**)&p, lp.p);
    }
};
```

Разница между *CComQIPtr* и *CComPtr* — в наличии второго параметра шаблона — *piid*, описывающего GUID интерфейса. У данного класса smart-указателя есть несколько конструкторов: конструктор по умолчанию, конструктор копии, конструктор с параметром типа указатель на произвольный интерфейс и конструктор с параметром типа указатель на *IUnknown*. Обратите внимание на последний: если разработчик создает объект и инициализирует его простым указателем на *IUnknown*, то вызывается *QueryInterface* с параметром шаблона типа GUID. Кроме того, заметьте, что присваивание указателя на *IUnknown* приводит к вызову *AtlComQIPtrAssign*. Думаем, вам понятно, что внутри этой функции происходит вызов *QueryInterface* с использованием параметра шаблона типа GUID.

Использование *CComQIPtr*

Перед вами пример использования *CComQIPtr* в коде COM-клиента:

```
void GetLottaPointers(ISomeInterface* pSomeInterface){
   HRESULT hr;
   CComQIPtr<IPersist, &IID_IPersist> persist;
   CComQIPtr<IDispatch, &IID_IDispatch> dispatch;
   CComPtr<IDataObject, &IID_IDataObject> dataobject;

   dispatch = pSomeInterface;   // неявный вызов QueryInterface
   persist = pSomeInterface;    // неявный вызов QueryInterface
   dataobject = pSomeInterface; // неявный вызов QueryInterface

   DoIt(persist, dispatch, dataobject);   // передача в функцию,
                                          // которой нужны IPersist*,
                                          // IDispatch* и IDataObject*

   // Деструкторы вызовут Release...
}
```

CComQIPtr полезен, если вам необходимо преобразование типов в стиле Java или Visual Basic. Кстати, в приведенном коде нет вызовов *QueryInterface* и *Release* — все они выполняются автоматически.

Проблемы применения smart-указателей в ATL

Smart-указатели порой весьма удобны (как в примере с *CComPtr*, что позволило избавиться от операторов *goto*). К сожалению, они не панацея, избавляющая программистов от всех проблем с подсчетом ссылок и управлением указателями. Smart-указатели просто переносят эти проблемы на другой уровень.

Одна из ситуаций, в которой надо соблюдать осторожность при использовании smart-указателей, заключается в преобразовании кода без таких указателей в код, использующий их. Проблема в том, что smart-указатели в ATL не скрывают вызовов *AddRef* и *Release*. Это значит, что вам следует понимать, как работают smart-указатели, а не следить за тем, как вызываются *AddRef* и *Release*. Например, представьте себе такой исходный текст:

```
void UseAnInterface(){
   IDispatch* pDispatch = NULL;
```

```
    HRESULT hr = GetTheObject(&pDispatch);
    if(SUCCEEDED(hr)) {
        DWORD dwTICount;
        pDispatch->GetTypeInfoCount(&dwTICount);
         pDispatch->Release();
    }
}
```

который преобразован в код с использованием smart-указателя:

```
void UseAnInterface() {
    CComPtr<IDispatch> dispatch = NULL;

    HRESULT hr = GetTheObject(&dispatch);
    if(SUCCEEDED(hr)) {
        DWORD dwTICount;
        dispatch->GetTypeInfoCount(&dwTICount);
        dispatch->Release();
    }
}
```

CComPtr и *CComQIPtr* не скрывают вызовы *AddRef* и *Release*, поэтому «слепое» преобразование создает проблему при освобождении smart-указателя *dispatch*. В приведенном коде *Release* будет вызываться дважды: первый раз явно при вызове *dispatch->Release()* и второй — неявно при выходе smart-указателя из области видимости.

Кроме того, в smart-указатели в ATL включен неявный оператор приведения типа, позволяющий присваивать их обычным указателям. В этом случае подсчет ссылок становится запутанным.

Вывод: хотя smart-указатели облегчают некоторые стороны программирования COM-клиента, они не являются «защитой от дурака». Чтобы безопасно применять smart-указатели, нужно хотя бы знать, как они работают.

Программирование сервера в ATL

Хотя значительная часть ATL относится к средствам разработки клиента (например, smart-указатели и BSTR-оболочки), основная часть ATL, к изучению которой мы приступаем, предназначена для поддержки создания COM-серверов. Сначала мы дадим обзор ATL, чтобы вам стала ясна общая картина. Затем мы заново реализуем модель космического корабля, чтобы исследовать ATL Simple Object Wizard и почувствовать, что значит писать COM-классы с использованием ATL.

ATL и COM-классы

Задача разработчика COM-класса — установить связь между таблицами функций и их реализациями и сделать необходимые вызовы *QueryInterface*, *AddRef* и *Release*. Как — дело ваше. Пользователям все равно, как вы этого добиваетесь. Пока мы видели два основных подхода — множественное наследование интерфейсов в обычном C++ и макросы и вложенные классы в MFC. Подход ATL для реализации COM-классов отличается от обоих.

Сравните подходы C++ и MFC. Вспомните, что разработка COM-классов на C++ подразумевает наследование класса C++ хотя бы от одного COM-интерфейса и написание всего кода такого класса. Далее вам придется вручную добавлять любые дополнительные возможности (например, поддержку *IDispatch* или агрегирование). Подход MFC к написанию COM-классов состоит в использовании макросов, определяющих вложенные классы (по одному на каждый интерфейс). MFC поддерживает *IDispatch* и агрегирование, поэтому вам не нужно много кода, чтобы добавить эти возможности. Однако очень трудно вставить в COM-класс новый интерфейс без написания вручную значительного объема кода. (Как вы видели в главе 22, поддержка COM в MFC основана на нескольких больших макросах.)

Подход ATL к созданию COM-класса состоит в создании класса C++, производного от нескольких классов-шаблонов. Однако в этих классах уже есть реализация *IUnknown*.

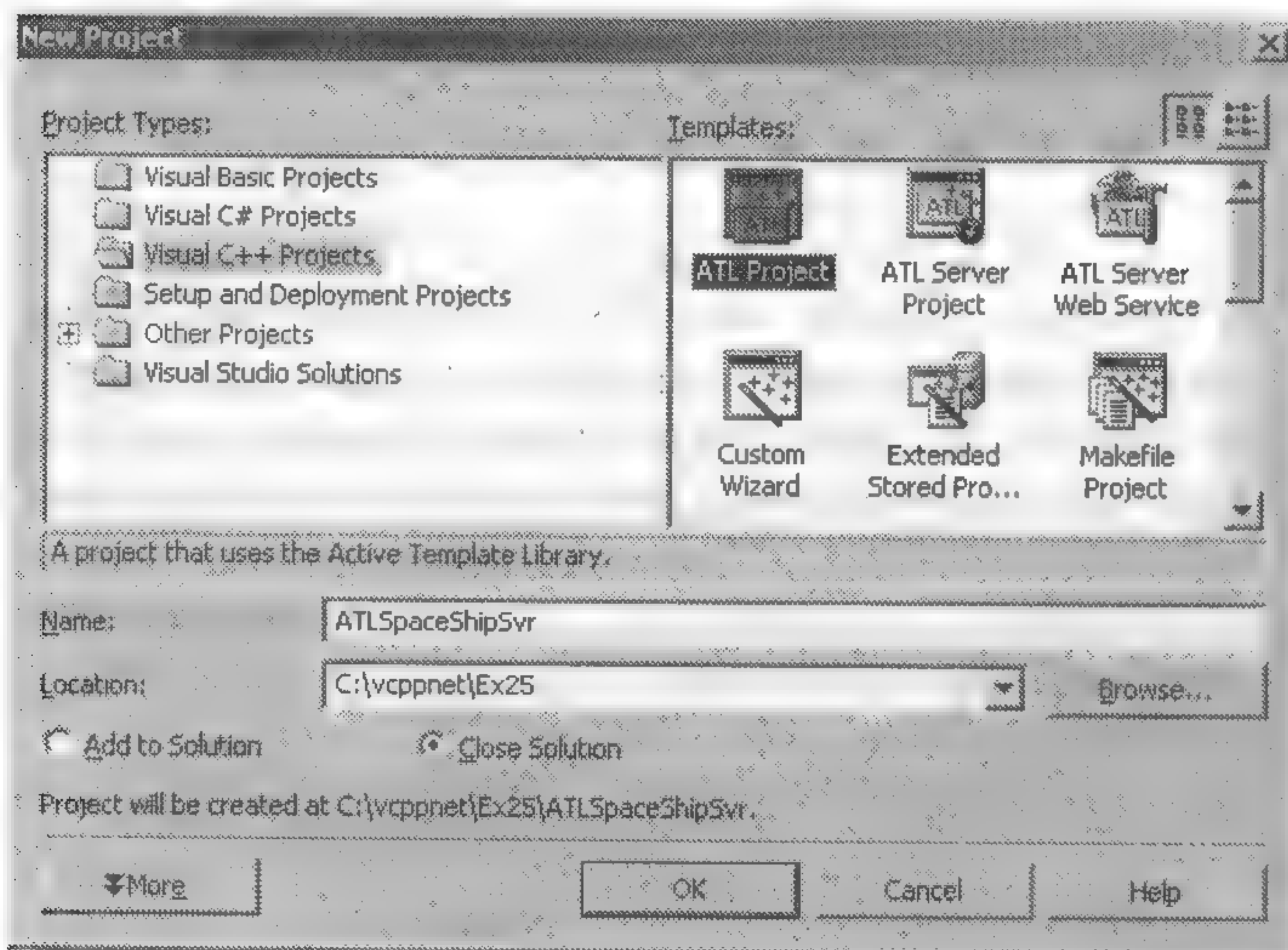


Рис. 25-1. *Выбор ATL Project в диалоговом окне New Project*

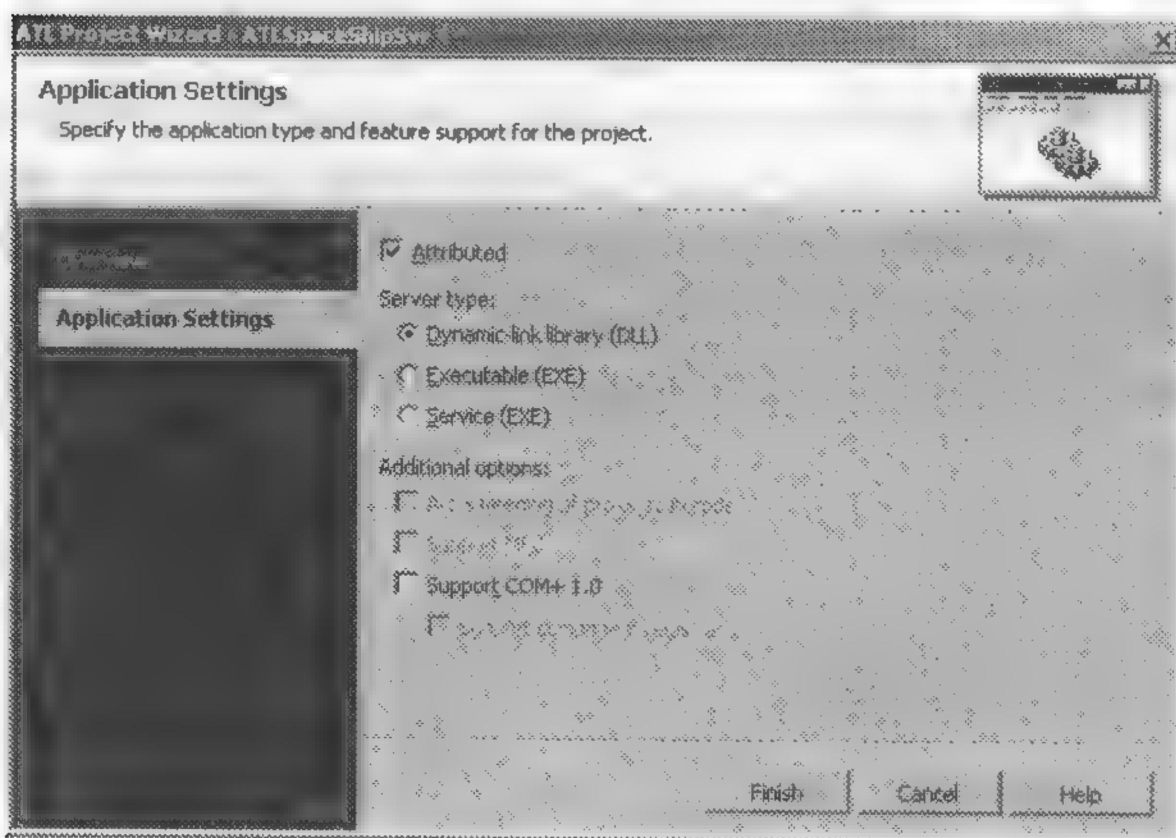


Рис. 25-2. *Страница Application Settings мастера ATL Project Wizard*

Давайте создадим модель космического корабля в виде COM-класса. Как всегда, начнем с выбора New Project в меню File. В открывшемся диалоговом окне New Project (рис. 25-1) в папке Visual C++ Projects выберите ATL Project. Назовите проект, скажем, **ATLSpaceShipSvr** и щелкните ОК. Откроется окно мастера ATL Project Wizard (рис. 25-2).

Параметры ATL-проекта

На странице Application Settings мастера ATL Project Wizard можно выбрать тип сервера: Dynamic Link Library (DLL), Executable (EXE) (исполняемый файл) или Service (EXE) (сервис). Если сбросить флажок Attributed и выбрать первый переключатель, в библиотеку можно включить код прокси/заглушки и использовать MFC в ATL-проекте. Есть также возможность обеспечить поддержку COM+ 1.0.

«Классическая» ATL и ATL с атрибутами

Мы уже упоминали флажок Attributed на странице Application Settings мастера ATL Project Wizard. Атрибуты — новая особенность Visual C++ .NET — призваны упростить программирование для COM и CLR-среды в .NET. Работа с атрибутами похожа на добавление сносок в исходный код — вы даете инструкции компилятору задействовать DLL провайдеров (provider DLL) для вставки кода или модификации сгенерированных объектных файлов. Атрибуты помогают Visual C++ .NET создавать IDL-файлы, интерфейсы, библиотеки типов и другие элементы COM. Атрибуты поддерживают мастера и страницы свойств среды Visual C++ .NET.

Если вы знакомы с языком IDL (Interface Definition Language), вы легко разберетесь с атрибутами. Многие из объявлений в IDL-описании преобразуются в атрибуты, которые попадают сразу в исходный, а не в IDL-код.

Язык C++ создан давно — еще до Windows он был популярным средством разработки. Как вы имели возможность убедиться при изучении COM, C++ — не лучшее средство создания DLL и компонентов, в частности из-за сложностей самого языка. Именно эту проблему призвана решить технология COM. Во многих отношениях COM «взяла» у C++ лучшие черты таблиц виртуальных функций, сопоставляемых реализации, и позволила создавать на C++ распространяемые DLL. В атрибутах сделан еще один шаг вперед

Атрибуты расширяют C++, не нарушая классической структуры языка. Они позволяют расширять возможности языка за счет *DLL провайдеров* (provider DLL). Основная задача атрибутов — упрощение программирования COM-компонентов. Их можно применять в большинстве конструкций языка C++, в том числе в классах и их членах — переменных и функциях.

Позже мы познакомимся поближе с программированием с применением «классической» ATL и ATL с атрибутами.

Выбор DLL в качестве типа сервера приводит к созданию необходимых элементов, вводящих DLL в среду COM. Среди них следующие стандартные функции COM: *DllGetClassObject*, *DllCanUnloadNow*, *DllRegisterServer* и *DllUnregisterServer*. Создаются также корректные механизмы управления временем жизни DLL.

При желании вы можете запустить DLL вне процесса в виде «суррогата», сбросив в мастере флажок Allow merging of proxy/stub code, который позволяет поместить все компоненты в один двоичный файл. (Обычно код прокси/заглушки поставляется как отдельная DLL.) Таким образом вы сможете поставлять только одну DLL. Если вам очень нужно включить MFC в свою DLL, установите флажок Support MFC. Это приведет к включению AfxWin.h и AfxDisp.h в файл StdAfx.h и компоновке проекта с текущей версией библиотеки импорта MFC. Использовать MFC очень удобно, и подчас не хочется «уходить» от этого, но не стоит забывать о возникающих при этом зависимостях. Если требуется поддержка сервисов COM+ 1.0 времени выполнения, установите флажок Support COM+ 1.0.

Если выбрать сервер в виде исполняемого файла, ATL Project Wizard создаст код, который компилируется в EXE-файл. Получаемый файл будет корректно регистрировать объекты классов в ОС с помощью *CoRegisterClassObject* и *CoRevokeClassObject*. В проект также включен код для управления временем жизни EXE-сервера. Наконец, если выбрать вариант Service (EXE), мастер ATL Project Wizard добавит код поддержки сервисов.

Применение ATL Project Wizard для написания «легковесного» сервера имеет ряд преимуществ. Проект сводит вместе весь исходный код и поддерживает необходимые инструкции для компиляции каждого файла.

Создание «классического» COM-класса

После создания COM-сервера можно переходить к добавлению в него COM-классов. К счастью, мастер ATL Simple Object Wizard (рис. 25-3) значительно облегчает выполнение этой задачи. Чтобы его вызвать, выберите Add Class в меню Project. В открывшемся окне выберите шаблон ATL Simple Object.

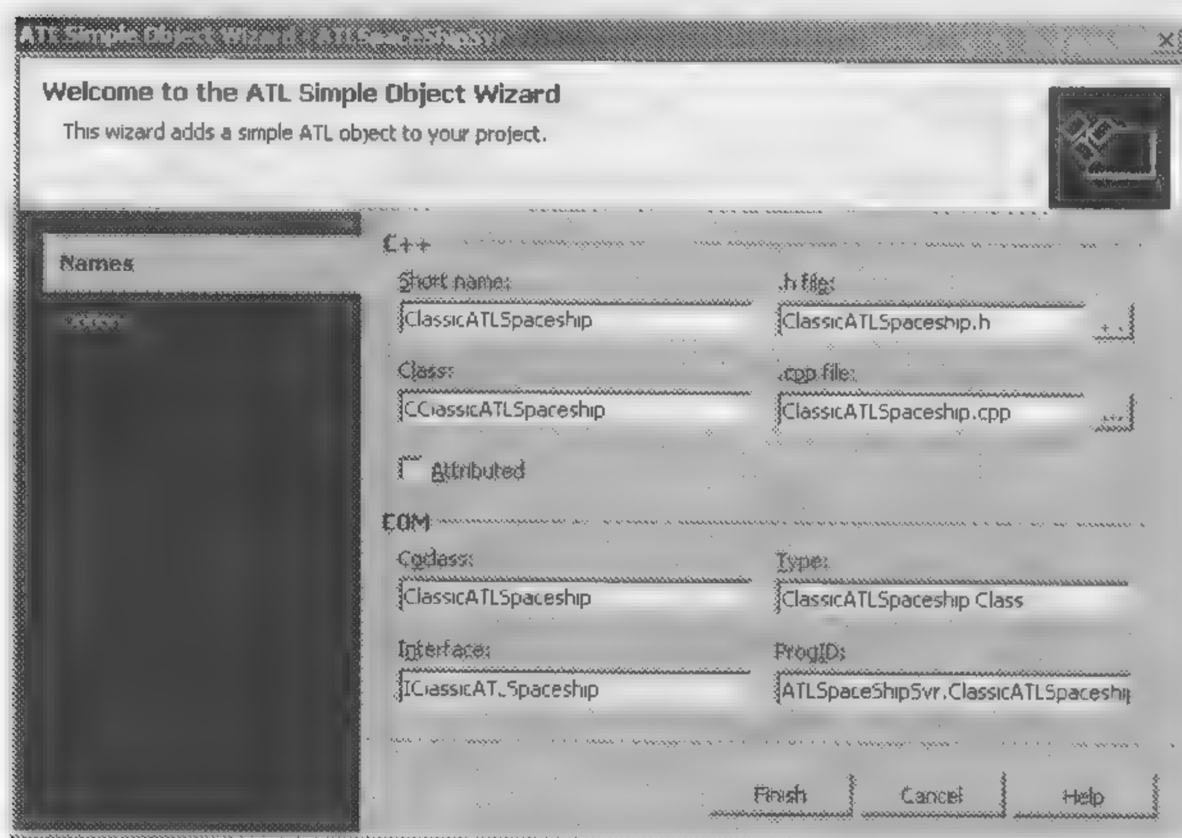


Рис. 25-3. *Создание COM-класса на основе ATL с помощью мастера ATL Simple Object Wizard*

Примечание Создавая классический COM-сервер не забудьте сбросить флажок Attributed на странице Application Settings мастера ATL Project Wizard.

При генерации нового объекта мастер ATL Simple Object Wizard добавляет в проект файл исходного кода и заголовочный файл на C++, содержащие реализацию и определение нового класса соответственно. Кроме того, он добавляет интерфейс в IDL-файл. Хотя мастер заботится о содержимом этого файла, вам все равно надо до некоторой степени понимать IDL, чтобы писать эффективные COM-интерфейсы (вскоре вы в этом убедитесь).

На странице Options мастера ATL Simple Object Wizard можно выбрать *модель потоков* (threading model), а также интерфейс — двойственный (основанный на *IDispatch*) или произвольный. Можно также выбрать поддержку агрегирования. Кроме того, мастер позволяет легко включить в класс интерфейс *ISupportErrorInfo* и *точки соединения* (connection point). Наконец, вы можете агрегировать в свой класс *маршалер свободных потоков* (free-threaded marshaler), выбрав в качестве модели потоков Both или Neutral.

Отделения и потоки

Для понимания COM важно представлять себе, что эта модель построена на *абстрагировании* (abstraction), т. е. на сокрытии от клиента максимума информации. Например, COM скрывает от клиента, является ли COM-класс *«потокобезопасным»* (thread-safe). Клиент должен иметь возможность использовать объект как он есть, не задумываясь о том, правильно ли объект организует доступ к самому себе, т. е. правильно ли он защищает свои внутренние данные. Для описания этого абстрагирования в COM введено понятие *отделение* (apartment).

Отделение определяет контекст выполнения (поток), которому принадлежат указатели на интерфейсы. Поток создает отделение вызывая одну из функций *CoInitialize*, *CoInitializeEx* или *OleInitialize*. Далее COM требует, чтобы все вызовы методов на основе указателя на интерфейс выполнялись внутри отделения, в котором инициализирован этот указатель (иначе говоря, из того же потока, в котором была вызвана *CoCreateInstance*). В COM определены два типа отделений: однопоточные и многопоточные. В первом разрешается только один поток, а во втором — несколько. В процессе может быть только одно многопоточное отделение, но произвольное число однопоточных. Отделение может содержать любое количество COM-объектов.

Однопоточное отделение гарантирует созданным в нем COM-объектам, что вызовы их методов синхронизируются через механизм *удаленных вызовов* (remoting layer)[1], а многопоточное — нет. Разницу между типами отделений можно пояснить так: создание COM-объекта внутри многопоточного отделения подобно размещению данных в глобальной области видимости с возможностью доступа нескольких потоков; а создание внутри однопоточного отделения — в области видимости одного потока. Отсюда следует, что COM-классы, которые предполагается разместить в многопоточных отделениях, должны быть потокобезопасными, а COM-классы, предпочитающие собственные отделения, могут не заботиться о совместном доступе к своим данным.

---

[1] Под удаленным вызовом здесь понимается любой вызов метода интерфейса в другом отделении. — *Прим. перев.*

Для COM-объекта, располагающегося в отдельном от клиента процессе, вызовы методов синхронизируются автоматически через механизм удаленных вызовов. Однако COM-объект внутри DLL может обеспечить собственную потокобезопасность (например, при помощи критических секций), вместо того чтобы полагаться на механизм удаленных вызовов. COM-класс сообщает о своей потокобезопасности через параметр реестра ThreadingModel в подразделе *CLSID* раздела *HKEY_CLASSES_ROOT*:

```
[HKCR\CLSID\{<некий GUID>}\InprocServer32]
@="C:\<некий сервер>.DLL"
ThreadingModel=<модель потоков>
```

где *<модель потоков>* принимает одно из 5 допустимых значений: *Single*, *Both*, *Free*, *Apartment* или *Neutral*, впрочем, значение может быть и не указано. ATL предоставляет поддержку для всех текущих потоковых моделей.

- *Single* или отсутствие значения означает, что класс может выполняться только в основном потоке (первом потоке, созданном клиентом).
- *Both* означает, что класс потокобезопасный и может выполняться как в однопоточном, так и в многопоточном отделении. Данное значение позволяет COM использовать для объекта тот же тип отделения, что и для клиента.
- *Free* означает, что класс потокобезопасный. Данное значение заставляет COM располагать объект внутри многопоточного отделения.
- *Apartment* означает, что класс не является потокобезопасным и должен размещаться в собственном однопоточном отделении.
- *Neutral* означает, что класс может размещаться в «потоконейтральном» («thread-neutral») отделении. Он подчиняется тем же правилам, что и многопоточный, но может работать в любом потоке.

В зависимости от выбранной потоковой модели мастер ATL Simple Object Wizard помещает в класс разный код. Так, при выборе модели Apartment класс будет производным от *CComObjectRootEx* с параметром шаблона *CComSingleThreadModel*:

```
class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                   &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
                    &IID_IClassicATLSpaceship,
                    &LIBID_SPACESHIPSVRLib>
{
:
};
```

Параметр шаблона *CComSingleThreadModel* приводит к более эффективным стандартным операциям инкремента/декремента в реализации *IUnknown*, поскольку доступ к классу синхронизируется автоматически. Кроме того, мастер ATL Simple Object Wizard генерирует код класса для добавления в реестр значения потоковой модели. При выборе в мастере модели Single класс также будет использовать *CComSingleThreadModel*, но значение параметра ThreadingModel в реестре останется пустым.

При выборе моделей Both или Free в классе будет применяться параметр шаблона *CComMultiThreadModel*, что приводит к использованию потокобезопасных Win32-функций инкремента/декремента *InterlockedIncrement* и *InterlockedDecrement*. Например, определение класса с моделью Free выглядит примерно так:

```
class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComMultiThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                   &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
                   &IID_IClassicATLSpaceship,
                   &LIBID_SPACESHIPSVRLib>
{
:
};
```

При выборе в мастере моделей Both или Free одноименная строка становится значением параметра ThreadingModel в реестре.

Точки соединения и *ISupportErrorInfo*

Добавить точки соединения к COM-классу очень просто. Установите флажок Connection points, и класс станет производным от *IConnectionPointImpl*; в него будет добавлена и пустая *карта соединений* (connection map). Чтобы добавить точки соединения (например, для поддержки событий) к классу, сделайте следующее.

1. Определите в IDL-файле интерфейс обратного вызова.
2. Используйте генератор прокси из ATL для создания класса-прокси.
3. Добавьте класс-прокси в COM-класс.
4. Добавьте точки соединения в *карту точек соединения* (connection point map).

ATL также содержит поддержку для *ISupportErrorInfo*. Этот интерфейс обеспечивает правильную маршрутизацию информации об ошибке по цепочке вызовов. Объекты OLE Automation, использующие интерфейсы с обработкой ошибок, должны реализовывать *ISupportErrorInfo*. Установка флажка ISupportErrorInfo в диалоговом окне ATL Simple Object Wizard приводит к тому, что ATL-класс становится производным от *ISupportErrorInfoImpl*.

Маршалер свободных потоков

Установка флажка Free-threaded marshaler приводит к агрегированию в классе маршалера свободных потоков COM. Как уже говорилось, эта возможность доступна только для объектов с потоковой моделью Both или Neutral. Это выполняется путем вызова *CoCreateFreeThreadedMarshaler* в функции *FinalConstruct*. Маршалер свободных потоков позволяет потокобезопасным объектам обойти стандартный маршалинг, который происходит при всяком вызове методов интерфейса между отделениями, и вызывать методы интерфейса в другом отделении так, будто они находятся в том же отделении. Тем самым значительно ускоряются вызовы между отеделениями. Маршалер свободных потоков осуществляет это, реализуя интерфейс *IMarshal*. Когда клиент запрашивает интерфейс у объекта, механизм удаленного вызова запрашивает *IMarshal* через *QueryInterface*. Если объект реали-

зует этот интерфейс (а в данном случае это так, поскольку мастер ATL Simple Object Wizard добавляет также элемент в карту интерфейсов класса, позволяющий *QueryInterface* обработать запросы от *IMarshal*) и есть запрос на маршалинг, то маршалер свободных потоков копирует указатель в пакет маршалинга. При этом клиент получает реальный указатель на объект и обращается к объекту напрямую, минуя прокси и заглушки. Конечно, при выборе флажка Free-threaded marshaler было бы лучше, чтобы все данные в вашем объекте были потокобезопасными, поэтому будьте очень осторожны при установке этого флажка.

Реализация класса *Spaceship* средствами классической ATL

Мы создадим класс космического корабля, используя значения по умолчанию, определенные в диалоговом окне ATL Simple Object Wizard. Например, класс будет иметь двойственный интерфейс, благодаря чему он будет, в частности, доступен из JScript на Web-странице. Кроме того, модель потоков класса — Apartment, поэтому COM будет контролировать основные вопросы синхронизации доступа. Единственное, что вам нужно указать мастеру ATL Simple Object Wizard, — это понятное имя. Введите нечто вроде **ClassicATLSpaceship** в поле ввода Short Name на странице Names.

Нет необходимости задавать другие параметры прямо сейчас. К примеру, не нужно устанавливать флажок Connection points, потому что мы рассмотрим соединения в следующей главе. Вы всегда можете добавить точки соединения, набрав нужный код вручную. Так выглядит определение класса, сгенерированное мастером.

```
// CClassicATLSpaceship

class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
      &IID_IClassicATLSpaceship,
      &LIBID_ATLSpaceShipSvrLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
:
};
```

Хотя в ATL довольно много COM-ориентированных классов C++, использованных для создания космического корабля базовых классов достаточно для понимания того, как работает ATL.

Большинство типовых COM-объектов, созданных с помощью ATL, наследует трем базовым классам: *CComObjectRoot*, *CComCoClass* и *IDispatch*. *CComObjectRoot* реализует *IUnknown* и поддерживает общность класса. Это значит, что в нем реализованы *AddRef*, *Release* и поддержка механизма ATL для *QueryInterface*. *CComCoClass* реализует объект класса и некоторую обработку ошибок. Реализуемый объект класса знает, как создавать объекты типа *CClassicATLSpaceship*. Наконец, в код, создаваемый мастером, включена реализация *IDispatch*, основанная на библиотеке ти-

пов, которая создается при компиляции IDL-файла. По умолчанию реализация базируется на двойственном интерфейсе, являющемся интерфейсом *IDispatch*, за которым следуют функции, определенные в IDL.

Как видите, для реализации COM-классов ATL применяется иначе, чем чистый C++. При использовании ATL главной частью проекта являются интерфейсы, описываемые в IDL-файле. Добавив функции к интерфейсам в IDL-коде, вы автоматически добавляете функции к конкретным классам, реализующим эти интерфейсы. Добавление будет автоматическим потому, что проекты настроены так, что компиляция IDL-файла создает заголовочный файл C++ с теми же функциями. Все, что остается вам сделать после добавления функций к интерфейсу, — это реализовать их в классе C++. Из IDL-файла также создается библиотека типов, благодаря чему COM-класс может реализовать *IDispatch*. Однако ATL полезна не только при реализации «легковесных» COM-объектов, но и является новым средством создания элементов управления ActiveX (см. главу 26).

Базовая архитектура ATL

Поэкспериментировав с ATL, вы увидите, насколько она упрощает реализацию COM-классов. Средства поддержки очень хороши: разрабатывать COM-классы инструментами Visual C++ .NET так же легко, как создавать программы на основе MFC. Вы просто используете ATL Project Wizard для создания сервера и ATL Simple Object Wizard — для нового создания ATL-класса. Как и в MFC, добавьте к интерфейсу определения новых функций с помощью Class View, а затем заполните кодом C++ сгенерированные функции. Код, создаваемый ATL Project Wizard, содержит необходимую реализацию класса, в том числе *IUnknown*, серверный модуль для COM-класса и объект класса с интерфейсом *IClassFactory*.

Описанный способ написания COM-объектов удобнее большинства других. Но что точно происходит, когда ATL Project Wizard генерирует код? Понимать, как работает ATL, важно, если вы собираетесь расширять свои COM-классы и серверы на основе ATL за пределы того, что предоставляют ATL Project Wizard и Class View. Например, ATL обеспечивает поддержку такой современной технологии, как *обособленные* (tear-off) интерфейсы. К сожалению, нет мастера или флажка для реализации обособленного интерфейса. Хотя в ATL есть поддержка обособленного интерфейса, вам придется часть работы проделать вручную. В этой ситуации очень помогает понимание ATL-реализации *IUnknown*.

Рассмотрим класс *CClassicATLSpaceship* более подробно:

```
// CClassicATLSpaceship
class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                  &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
      &IID_IClassicATLSpaceship,
      &LIBID_ATLSpaceShipSvrLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
   CClassicATLSpaceship()
```

```
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_CLASSICATLSPACESHIP)

BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }
    void FinalRelease()
    {
    }
public:
};

OBJECT_ENTRY_AUTO(__uuidof(ClassicATLSpaceship), CClassicATLSpaceship)
```

Хотя это обычный исходный код на C++, он во многом отличается от кода реализации COM-объектов. Например, обычно COM-класс наследует интерфейсам напрямую, а здесь он производный от шаблонов. Кроме того, в этом классе C++ используется несколько, на первый взгляд, странных макросов. По мере изучения кода вы увидите ATL-реализацию *IUnknown*, узнаете о методе предотвращения «распухания» Vtbl и познакомитесь с необычным применением шаблонов. Начнем с первого макроса, созданного мастером: *ATL_NO_VTABLE*.

Предотвращение «распухания» Vtbl

COM-интерфейсы легко описываются на C++ как чисто абстрактные базовые классы. Создание COM-классов путем множественного наследования (есть и другие способы) состоит просто в добавлении интерфейса в качестве базового класса и реализации всех его функций. Конечно, это означает, что в выделяемой для вашего COM-сервера памяти размещается значительных размеров *таблица виртуальных функций* (Vtbl) для каждого реализованного интерфейса. Это не страшно, если у вас немного интерфейсов и иерархия классов C++ не очень глубока. Однако в таком способе реализации интерфейсов размер таблицы имеет тенденцию к росту по мере добавления интерфейсов и углубления иерархии. ATL позволяет уменьшить потери памяти, вызываемые множеством виртуальных функций, — определяется символ:

```
#define ATL_NO_VTABLE __declspec(novtable)
```

Макрос *ATL_NO_VTABLE* предотвращает инициализацию в конструкторе объекта vtbl, тем самым удаляя эту таблицу и все указатели на виртуальные функции это-

го класса из процесса компоновки. Это отчасти уменьшает размер COM-сервера до размеров последнего в иерархии класса, не использующего директиву declspec novtable. Выигрыш заметнее в развитых иерархиях классов. Но внимание: небезопасно вызывать виртуальные функции в конструкторе любого объекта, объявленного с этой директивой, поскольку указатель на таблицу виртуальных функций не инициализирован.

Вторая строка в объявлении класса говорит о том, что *CClassicATLSpaceship* наследует классу *CComObjectRootEx*. Вот мы и добрались до ATL-версии *IUnknown*.

ATL-версия *IUnknown*: *CComObjectRootEx*

CComObjectRootEx не венчает иерархию ATL, но он очень близок к вершине. Настоящий базовый класс для COM-объекта в ATL — это *CComObjectRootBase*. (Определения обоих классов находятся в AtlCom.h.) При изучении кода *CComObjectRootBase* можно обнаружить, что ожидается от базового COM-класса на C++. Класс содержит переменную-член *m_dwRef* типа *DWORD* для подсчета ссылок. В нем есть методы *OuterAddRef*, *OuterRelease* и *OuterQueryInterface* для поддержки агрегирования и обособленных интерфейсов. Изучение *CComObjectRootEx* позволяет обнаружить *InternalAddRef*, *InternalRelease* и *InternalQueryInterface* для реального выполнения подсчета ссылок и механизмы реализации *QueryInterface* для экземпляров классов с ***объектной общностью*** (object identity).

Из определения класса *CClassicATLSpaceship* видно, что он происходит от *CComObjectRootEx* и что последний является параметризованным классом-шаблоном. Определение *CComObjectRootEx* таково:

```
template <class ThreadModel>
class CComObjectRootEx : public CComObjectRootBase
{
public:
    typedef ThreadModel _ThreadModel;
    typedef _ThreadModel::AutoCriticalSection _CritSec;
    typedef CComObjectLockT<_ThreadModel> ObjectLock;

    ULONG InternalAddRef()
    {
        ATLASSERT(m_dwRef != -1L);
        return _ThreadModel::Increment(&m_dwRef);
    }
    ULONG InternalRelease()
    {
#ifdef _DEBUG
        LONG nRef = _ThreadModel::Decrement(&m_dwRef);
        if (nRef < -(LONG_MAX / 2))
        {
            ATLASSERT(0 && _T("Release called on a pointer "
                        "that has already been released"));
        }
        return nRef;
#else
```

```
        return _ThreadModel::Decrement(&m_dwRef);
#endif
    }

    void Lock() {m_critsec.Lock();}
    void Unlock() {m_critsec.Unlock();}
private:
    _CritSec m_critsec;
};
```

CComObjectRootEx — это класс-шаблон, который зависит от класса потоковой модели, передаваемого как параметр шаблона. Фактически ATL поддерживает несколько моделей потоков: *однопотоковые отеделения* (Single-Threaded Apartment, STA), *многопотоковые отделения* (Multi-Threaded Apartments, MTA) и *свободные потоки* (Free Threading). Для определения модели потоков в проекте может быть один из трех символов препроцессора: *_ATL_SINGLE_THREADED*, *_ATL_APARTMENT_THREADED* или *_ATL_FREE_THREADED*.

Определение *_ATL_SINGLE_THREADED* в файле Stdafx.h задает поддержку только одного потока на основе STA. Этот вариант пригоден для внешних серверов, не создающих дополнительных потоков. Поскольку сервер поддерживает только один поток, переменные глобального состояния ATL можно не защищать критическим секциями, вследствие чего сервер работает эффективнее. Недостаток — сервер поддерживает только один поток. Определение *_ATL_APARTMENT_ THREADED* обеспечивает поддержку нескольких потоков на основе STA. Это полезно для внутренних серверов с моделью отделений (т. е. у которых в реестре задана пара «параметр — значение» *ThreadingMode=Apartment*). Поскольку сервер этой потоковой модели может поддерживать несколько потоков, ATL защищает свои переменные глобального состояния при помощи критических секций. Наконец, определение *_ATL_FREE_THREADED* создает сервер, совместимый с любой потоковой средой. Поэтому ATL предохраняет переменные глобального состояния критическими секциями, плюс каждый объект имеет собственные критические секции для защиты своих данных.

Перечисленные символы препроцессора просто определяют потоковый класс, который передается в *CComObjectRootEx* в качестве параметра шаблона. ATL предоставляет три класса потоковых моделей, обеспечивающих поддержку наиболее эффективного, но потокобезопасного поведения COM-классов внутри каждого из трех перечисленных контекстов. Эти классы — *CComMultiThreadModelNoCS*, *CComMultiThreadModel* и *CComSingleThreadModel*.

```
class CComMultiThreadModelNoCS
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw()
        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p) throw()
        {return InterlockedDecrement(p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
```

```
};

class CComMultiThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw()
        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p) throw()
        {return InterlockedDecrement(p);}
    typedef CComAutoCriticalSection AutoCriticalSection;
    typedef CComCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
};

class CComSingleThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw() {return ++(*p);}
    static ULONG WINAPI Decrement(LPLONG p) throw() {return -(*p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComSingleThreadModel ThreadModelNoCS;
};
```

Заметьте: в каждом классе две статические функции, *Increment* и *Decrement*, и ряд синонимов (alias) для критических секций.

CComMultiThreadModel и *CComMultiThreadModelNoCS* реализуют *Increment* и *Decrement* через вызов потокобезопасных Win32-функций *InterlockedIncrement* и *InterlockedDecrement*. *CComSingleThreadModel* реализует эти же функции при помощи обычных операторов «++» и «--».

Помимо различия в реализации инкремента/декремента, три модели потоков по-разному используют критические секции. ATL предоставляет две оболочки критических секций: *CComCriticalSection* (это обычная обертка вокруг Win32 API критической секции) и *CComAutoCriticalSection* (то же, что и *CComCriticalSection*, но с автоматической инициализацией/очисткой). ATL также определяет класс «фиктивной» критической секции, имеющей те же методы, что и другие классы критических секций, но ничего не делающей. Как видно из определений классов, в *CComMultiThreadModel* используются настоящие критические секции, тогда как *CComMultiThreadModelNoCS* и *CComSingleThreadModel* — фиктивные, ничего не делающие критические секции.

Теперь более понятно минимальное определение класса ATL. При всяком указании класса *CComObjectRootEx* в него передается класс модели потоков. *CClassicATLSpaceship* определен с помощью класса *CComSingleThreadModel*, поэтому он использует методы этого класса для инкремента/декремента, а также фиктивные критические секции. Следовательно, *CClassicATLSpaceship* ведет себя наиболее эффективно, так как не должен заботиться о защите данных. Однако вы не привязаны к этой модели. Если хотите сделать *CClassicATLSpaceship* потокобезопасным, просто укажите *CComMultiThreadModel* в качестве параметра шаблона для

CComObjectRootEx. Тогда вызовы *AddRef* и *Release* автоматически сопоставляются корректным функциям *Increment* и *Decrement*.

ATL и *QueryInterface*

Похоже, что в реализации *QueryInterface* ATL берет пример с MFC, так как тоже использует поисковую таблицу. Взгляните в середину определения *CClassicATLSpaceship* — вы увидите состоящую из макросов *карту интерфейсов* (interface map). Карты интерфейсов в ATL формируют механизм реализации *QueryInterface*.

Клиенты используют *QueryInterface* для произвольного расширения связи с объектом. Это значит, что, когда клиенту нужен новый интерфейс, он вызывает *QueryInterface* для существующего интерфейса. Если объект реализует искомый интерфейс, он возвращает его клиенту, нет — возвращается ошибка, обозначающая, что интерфейс не найден.

Традиционные реализации *QueryInterface* обычно состоят из длинных операторов if-then. Так, для COM-класса с множественным наследованием эта функция может выглядеть так:

```
class CClassicATLSpaceship: public IDispatch,
                            IClassicATLSpaceship {
   HRESULT QueryInterface(RIID riid,
                          void** ppv) {
      if(riid == IID_IDispatch)
         *ppv = (IDispatch*) this;
      else if(riid == IID_IClassicATLSpaceship ||
              riid == IID_IUnknown)
         *ppv = (IClassicATLSpaceship *) this;
      else {
         *ppv = 0;
         return E_NOINTERFACE;
      }

      ((IUnknown*)(*ppv))->AddRef();
      return NOERROR;
   }
   // AddRef, Release и другие функции
};
```

Как вы уже видели, ATL использует поисковую таблицу вместо обычного оператора if-then. Эта таблица начинается с макроса *BEGIN_COM_MAP*. Ниже приведено его полное определение.

```
#define BEGIN_COM_MAP(x) public: \
   typedef x _ComMapClass; \
   static HRESULT WINAPI _Cache(void* pv, \
      REFIID iid, void** ppvObject, \
      DWORD_PTR dw) throw() \
   { \
      _ComMapClass* p = (_ComMapClass*)pv; \
      p->Lock(); \
      HRESULT hRes = \
```

```
         ATL::CComObjectRootBase::_Cache(pv, iid, ppvObject, dw); \
      p->Unlock(); \
      return hRes; \
   } \
   IUnknown* _GetRawUnknown() throw() \
   { ATLASSERT(_GetEntries()[0].pFunc == _ATL_SIMPLEMAPENTRY); \
      return (IUnknown*)((INT_PTR)this+_GetEntries()->dw); } \
   _ATL_DECLARE_GET_UNKNOWN(x) \
   HRESULT _InternalQueryInterface(REFIID iid, \
      void** ppvObject) throw() \
   { return InternalQueryInterface(this, \
      _GetEntries(), iid, ppvObject); } \
   const static ATL::_ATL_INTMAP_ENTRY* WINAPI _GetEntries() \
      throw() { \
   static const ATL::_ATL_INTMAP_ENTRY _entries[] = \
      { DEBUG_QI_ENTRY(x)
```

Каждый класс, в котором для реализации *IUnknown* применяется ATL, должен задать карту интерфейсов для реализации *InternalQueryInterface*. Карта интерфейсов в ATL состоит из структур, содержащих тройку: идентификатор интерфейса (GUID)/переменная типаDWORD/указатель на функцию. Далее показан тип *_ATL_INTMAP_ENTRY*, поддерживающий эту тройку.

```
struct _ATL_INTMAP_ENTRY
{
   const IID* piid;    // Идентификатор интерфейса (IID)
   DWORD_PTR dw;
   _ATL_CREATORARGFUNC* pFunc; //NULL:end, 1:offset, n:ptr
};
```

Первый элемент — это идентификатор интерфейса (GUID), второй — признак действия, выполняемого при запросе интерфейса. Третий интерпретируется по-разному. Если *pFunc* равно константе *_ATL_SIMPLEMAPENTRY* (значение 1), то *dw* — это смещение внутри объекта. Если *pFunc* не 0 и не 1, то указывает на функцию, которая вызывается при запросе интерфейса. Если *pFunc* равно *NULL*, то *dw* означает конец поисковой таблицы *QueryInterface*.

Заметьте: в *CClassicATLSpaceship* используется макрос *COM_INTERFACE_ENTRY*. Это элемент карты для обычных интерфейсов. Вот этот макрос полностью:

```
#define offsetofclass(base, derived) \
   ((DWORD_PTR) \
   (static_cast<base*>((derived*)_ATL_PACKING))-_ATL_PACKING)

#define COM_INTERFACE_ENTRY(x) \
   {&_ATL_IIDOF(x), \
   offsetofclass(x, _ComMapClass), \
   _ATL_SIMPLEMAPENTRY}
```

COM_INTERFACE_ENTRY вносит значение идентификатора интерфейса в структуру *_ATL_INTMAP_ ENTRY*. Кроме того, обратите внимание, как *offsetofclass* преобразует указатель *this* в нужный интерфейс и помещает полученное значение в поле

dw. Наконец, *COM_INTERFACE_ ENTRY* помещает в последнее поле значение *_ATL_SIMPLEMAPENTRY*, чтобы указать на то, что *dw* — смещение внутри класса.

Например, карта интерфейсов для *CClassicATLSpaceship* после обработки препроцессором выглядит так:

```
const static _ATL_INTMAP_ENTRY* __stdcall _GetEntries() {
    static const _ATL_INTMAP_ENTRY _entries[] = {
        {&IID_IClassicATLSpaceship,
         ((DWORD)(static_cast<IClassicATLSpaceship*>
             ((_ComMapClass*)8))-8),
        ((_ATL_CREATORARGFUNC*)1)},
        {&IID_IDispatch,
         ((DWORD)(static_cast<IDispatch*>((_ComMapClass*)8))-8),
         ((_ATL_CREATORARGFUNC*)1)},
         {0, 0, 0}
    };
    return _entries;
}
```

В настоящий момент класс *CClassicATLSpaceship* поддерживает два интерфейса *IClassicATLSpaceship* и *IDispatch*, поэтому в карте два элемента.

Реализация *InternalQueryInterface* в *CComObjectRootEx* получает функцию *_GetEntries* в качестве второго параметра. В этой реализации используется глобальная ATL-функция *AtlInternalQueryInterface* для поиска интерфейса в карте путем ее простого просмотра.

Помимо *COM_INTERFACE_ENTRY*, в состав ATL входят еще 16 макросов, реализующих различные способы композиции объектов: от обособленных интерфейсов до агрегирования. Далее вы увидите, как расширить интерфейс *IClassicATLSpaceship* путем добавления двух других интерфейсов — *IMotion* и *IVisual*. Вы также узнаете о странном «звере» в мире COM — *двойственном интерфейсе* (dual interface).

Космический корабль учится летать

Итак, некоторый ATL-код у вас есть — что дальше? Поскольку речь идет о COM, то начать нужно с IDL-файла. Это может быть новая, незнакомая вам сторона разработки ПО. Помните: сейчас такое время, когда распространение и интеграция ПО становятся очень важными. Ранее вы могли разобраться в деталях классов C++ и «подцеплять» их в свой проект, поскольку вы (как разработчик) видели всю картину. Однако в компонентных технологиях (вроде COM) все иначе: вы не видите всей картины целиком. Очень часто у вас есть только компонент без его исходного кода. Единственный способ узнать, что он может делать, — вызвать предоставляемые им интерфейсы.

Имейте в виду, что разработчики ПО используют массу средств, не только C++. Компоненты программируют на Visual Basic, Java, Delphi и С. COM «сглаживает» все углы при интеграции программных блоков, созданных этими программными средствами. Кроме того, удаленное выполнение ПО (как внешнего процесса на этой же или другой машине) требует межпроцессного взаимодействия. Для решения этих проблем и создан язык определения интерфейсов IDL (Interface Definition Language,). Вот IDL-файл, созданный мастером для нового класса *Spaceship*:

```
import "oaidl.idl";
import "ocidl.idl";

[
   object,
   uuid(45896187-46FF-4A07-A9DC-557377380535),
   dual,
   nonextensible,
   helpstring("IClassicATLSpaceship Interface"),
   pointer_default(unique)
]
interface IClassicATLSpaceship : IDispatch{
};
[
   uuid(F5FD4043-22AE-470D-8C43-1AC904D2E8E0),
   version(1.0),
   helpstring("ATLSpaceShipSvr 1.0 Type Library")
]
library ATLSpaceShipSvrLib
{
   importlib("stdole2.tlb");
   [
      uuid(E485E21E-A23C-413F-A93B-909318565113),
      helpstring("ClassicATLSpaceship Class")
   ]
   coclass ClassicATLSpaceship
   {
      [default] interface IClassicATLSpaceship;
   };
};
```

Ключевая концепция IDL состоит в том, что это чисто *декларативный* (declarative) язык. Он определяет, как клиенты могут общаться с объектом. Помните: этот код обрабатывается MIDL-компилятором для получения чисто абстрактного базового класса, используемого клиентами на C++, и библиотеки типов, используемой клиентами на Visual Basic, Java и других языках. Если вы способны разобраться в обычном коде на C, то сможете понять и IDL. Образно говоря, IDL — это C со сносками. В соответствии с правилами синтаксиса IDL атрибуты всегда предшествуют элементу, который они описывают. Так, атрибуты предшествуют объявлениям интерфейсов, библиотек и параметров методов.

Заметьте: IDL-файл начинается с импорта файлов Oaidl.idl и Ocidl.idl, который подобен включению Windows.h в файлы на C или C++. Указанные IDL-файлы содержат определения всех базовых элементов COM (в том числе определения *IUnknown* и *IDispatch*).

За операторами *import* идет открывающая квадратная скобка ([). В IDL в квадратных скобках всегда помещаются атрибуты. Первый элемент этого IDL-файла — интерфейс *IClassicATLSpaceship*. Однако до описания этого интерфейса надо указать его атрибуты. Например, дать ему имя (GUID), указать MIDL-компилятору, что это COM-интерфейс, а не интерфейс для стандартного RPC, а также что это

двойственный интерфейс (о двойственных интерфейсах см. ниже). Далее идет сам интерфейс. Заметьте, что его строение очень похоже на обычную структуру языка С.

После описания интерфейсов полезно собрать эту информацию в библиотеке типов, что делается в следующем разделе IDL-файла. Кстати, библиотека типов также начинается с открывающей квадратной скобки, которая обозначает следующие за ней атрибуты. Как всегда, библиотека типов — как и всякий «независимый» элемент в COM — требует имя (GUID). Оператор описания библиотеки сообщает MIDL-компилятору, что эта библиотека включает COM-класс *ClassicATLSpaceship* и что клиенты этого класса могут запросить интерфейс *IClassicATLSpaceship*.

Добавление методов в интерфейс

В теперешнем виде интерфейс *IClassicATLSpaceship* неприменим — ему нужны один-два метода. Давайте их добавим. При добавлении свойств Automation в COM-классы на основе MFC мы использовали окно Class View. То же самое мы сделаем в ATL. Заметьте: *CClassicATLSpaceship* наследует *IClassicATLSpaceship*. Естественно, что *IClassicATLSpaceship* — COM-интерфейс. Двойной щелчок *IClassicATLSpaceship* в окне Class View отобразит в окне редактора соответствующий раздел IDL-файла.

В данный момент вы можете изменить COM-интерфейс прямо в IDL-файле. Если вы добавите таким образом функции и методы, вам придется обратиться к файлам ClassicATLSpaceship.h и ClassicATLSpaceship.cpp и ввести методы вручную. Продуктивнее добавлять функции в интерфейсы через Class View с помощью мастера Add Method Wizard (рис. 25-4). Для этого в окне Class View просто щелкните интерфейс правой кнопкой. Вы увидите в контекстном меню команды Add Method и Add Property. Давайте добавим метод *CallStarFleet* (вызвать звездный флот).

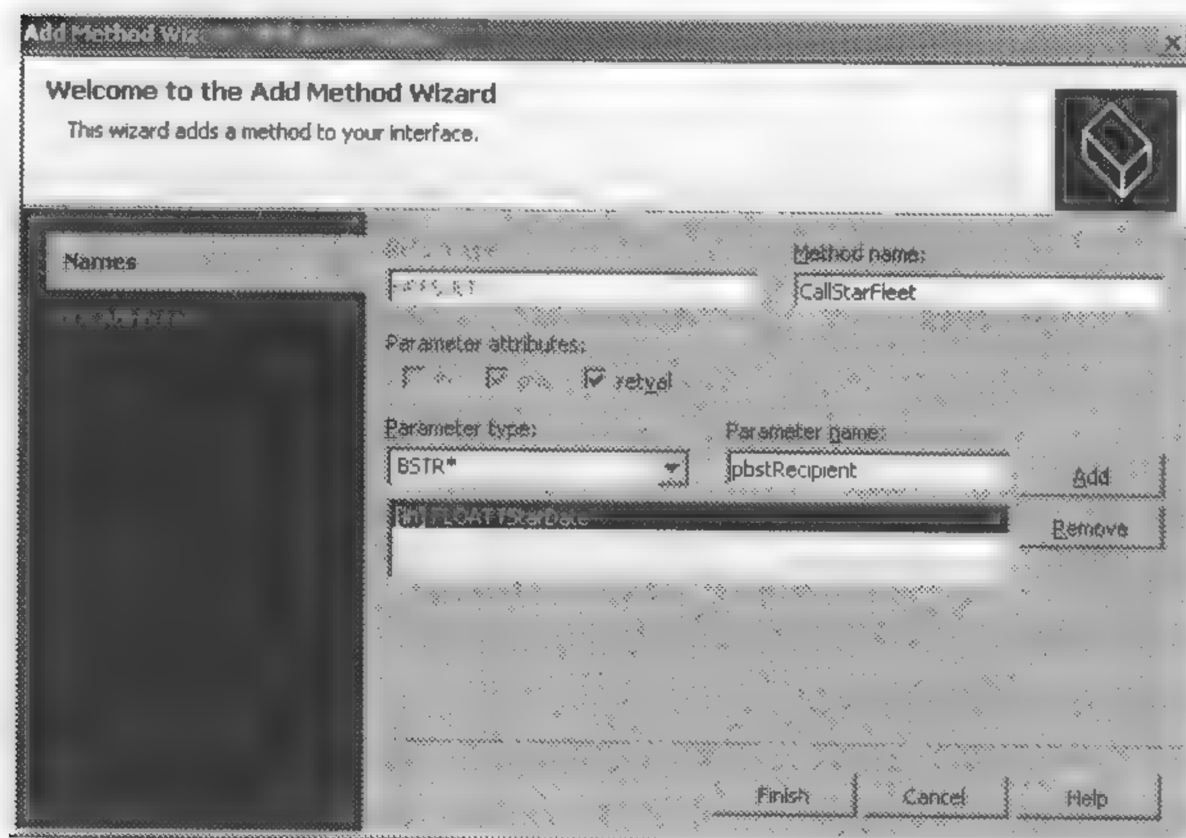


Рис. 25-4. *Добавление метода в интерфейс*

Чтобы добавить метод, введите его имя в поле ввода Method Name и укажите его параметры в полях ввода Parameter name и Parameter type. Здесь нужно объяснить одну деталь относительно IDL.

Помните, что цель IDL — предоставить однозначную информацию о вызове методов. В стандартном мире C++ вы часто имеете дело с неоднозначными вещами, например, с массивами неизвестной длины, что не проблематично, посколь-

ку вызывающий и вызываемый используют один и тот же фрейм стека и всегда найдется нужное количество памяти. Но так как вызовы методов могут происходить по сети, то важно точно сообщить механизму удаленных вызовов, что собой представляет COM-интерфейс. Сделать это можно, добавив атрибуты к параметрам метода (опять квадратные скобки).

У метода *CallStarFleet* два параметра: число с плавающей запятой (обозначает дату) и BSTR (описывает получателя). В методе описываются направления передачи параметров. Дата передается в метод при его вызове, что обозначается атрибутом [in]. Строка, описывающая получатель, передается обратно в виде указателя на BSTR. Атрибут [out] означает направление передачи параметра от объекта к клиенту. Атрибут [retval] означает, что результат вызова метода можно присвоить переменной в языке высокого уровня, поддерживающем такую возможность.

Двойственные интерфейсы

В главе 23 вы видели интерфейс *IDispatch*, позволяющий представить функциональность объекта (на двоичном уровне) для языков высокого уровня, подобных JScript, которые не имеют представления о виртуальных таблицах (vtbl). Чтобы *IDispatch* заработал, клиенту нужно выполнить ряд хитроумных действий перед вызовом *Invoke*. Сначала клиент получает идентификаторы вызова[1], затем настраивает аргументы типа VARIANT. На стороне объекта декодируются все VARIANT-параметры, проверяется их корректность, затем они размещаются в стеке, и вызывается функция. Ясно, что это сложная и длительная работа.

Если вы пишете COM-объект и предполагаете, что одна часть клиентов будет использовать языки сценариев, а другая — языки, подобные C++, то вы получите дилемму: или вы включаете интерфейс *IDispatch*, или лишаетесь клиентов на языках сценариев. Если вы предоставите только *IDispatch*, то доступ к объекту из C++ будет очень неудобен. Конечно, можно предоставить доступ как через *IDispatch*, так и через собственный интерфейс, но это требует дополнительного программирования. Данную проблему решают двойственные интерфейсы.

Двойственный интерфейс (dual interface) — это просто *IDispatch* с добавленными в конец функциями. Например, это «двойственная версия» интерфейса *IMotion*:

```
interface IMotion : public IDispatch {
   virtual HRESULT Fly() = 0;
   virtual HRESULT GetPosition() = 0;
};
```

Так как *IMotion* происходит от *IDispatch*, то первые семь функций *IMotion* — это функции *IDispatch*. Клиенты, понимающие только *IDispatch* (например, JScript), считают этот интерфейс одной из разновидностей *IDispatch* и для вызова функций передают их DISPID в *Invoke*. Клиенты, понимающие произвольные интерфейсы на основе vtbl, воспринимают этот интерфейс целиком, игнорируют четыре функции в середине (относящиеся непосредственно к *IDispatch*) и работа-

[1] Автор имеет в виду disp-идентификаторы методов и свойств. — *Прим. перев.*

ют с первыми тремя (от *IUnknown*) и последними двумя (от самого *IMotion*) функциями. На рис. 25-5 показан формат vtbl для *IMotion*.

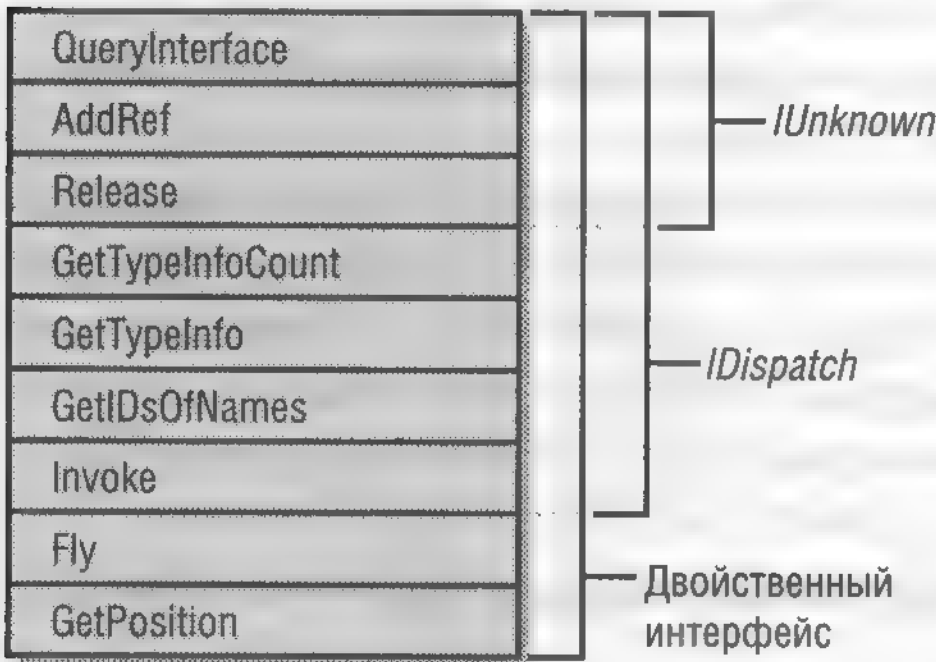


Рис. 25-5. *Структура двойственного интерфейса*

Во многих реализациях на C++ выполняется загрузка библиотеки типов и делегирование интерфейсу *ITypeInfo* неприятной задачи по реализации вызовов *Invoke* и *GetIDsOfNames*. Подробности см. в книгах Брокшмидта и Роджерсона.

ATL и *IDispatch*

В ATL реализация *IDispatch* находится в классе *IDispatchImpl* и делегирует вызовы библиотеке типов. Объекты, реализующие двойственный интерфейс, должны включать шаблон *IDispatchImpl* в число базовых классов:

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship, &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship, &IID_IClassicATLSpaceship,
                    &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IVisual, &IID_IVisual,
                    &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                    &LIBID_SPACESHIPSVRLib>
{
:
};
```

Помимо добавления класса-шаблона *IDispatchImpl* в список базовых, у объекта должны быть элементы таблицы интерфейсов для двойственного интерфейса и для *IDispacth*, чтобы *QueryInterface* работала правильно.

```
BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

Как видите, аргументами класса-шаблона *IDispatchImpl* являются сам двойственный интерфейс, его GUID и GUID библиотеки типов, хранящей всю информацию об интерфейсе. Кроме того, есть ряд необязательных параметров, не показанных на рис. 25-5, в том числе номер версии библиотеки типов [причем как сокращен-

ной (minor), так и полной (major) версии] и класс для управления информацией о типах. По умолчанию используется ATL-класс *CComTypeInfoHolder*.

В большинстве обычных реализаций *IDispatch* на C++ в конструкторе класса выполняется вызов *LoadTypeLib* и *ITypeLib::GetTypeInfoOfGuid* с последующим сохранением указателя на *ITypeInfo* на весь период существования класса. В ATL реализация сделана чуть иначе — на основе класса *CComTypeInfoHolder*. В этом классе есть переменная-член типа указатель на *ITypeInfo* и «обертки» основных функций *IDispatch*: *GetIDsOfNames* и *Invoke*.

Клиенты запрашивают двойственный интерфейс, вызывая *QueryInterface* для *IID_IClassicATLSpaceship*. (Этот же интерфейс клиент может получить, вызвав *QueryInterface* для *IDispatch*.) При обращении к *CallStarFleet* эта функция будет вызвана напрямую (как для любого другого COM-интерфейса).

Когда клиент вызывает *IDispatch::Invoke*, вызов передается в функцию *Invoke* класса *IDispatchImpl*, которая делегирует вызов в *Invoke* класса *CComTypeInfoHolder*. В этом классе загрузка библиотеки типов (вызов *LoadTypeLib*) откладывается до первого обращения к *Invoke* или *GetIDsOfNames*. Для обращения к библиотеке типов на основе информации в реестре (используя GUID и номер версии, переданные как параметры шаблона) в *CComTypeInfoHolder* есть функция-член *GetTI*. Затем *CComTypeInfoHolder* вызывает *ITypeLib::GetTypeInfo* для получения информации об интерфейсе (указатель на *ITypeInfo*). Далее все вызовы делегируются этому интерфейсу. Реализация *GetIDsOfNames* в *IDispatchImpl* выполнена аналогично.

Интерфейсы *IMotion* и *IVisual*

Чтобы сделать наш COM-класс похожим на другие его версии (C++ и MFC, описанные в главе 22), нужно добавить в него и в проект интерфейсы *IMotion* и *IVisible*. Увы, мастера создания интерфейсов в Visual Studio .NET нет. Для этого сначала можно задействовать ATL Simple Object Wizard, чтобы создать объект. Альтернативный путь — кодировать вручную. Откройте IDL-файл, поместите курсор в начале, где-то после операторов *#import*, но до оператора *library*, и введите определения интерфейсов, как описано далее.

Зная IDL, вы автоматически должны начать описание интерфейса с открывающей квадратной скобки ([). Помните, что в IDL отдельные элементы имеют атрибуты, и один из важнейших — имя (GUID). Кроме того, очень важно для интерфейса иметь атрибут *object*, обозначающий принадлежность к COM (а не к обычному RPC). Возможно, вы захотите сделать интерфейсы двойственными. В этом случае добавьте к атрибутам ключевое слово *dual*, которое приведет к внесению в реестр соответствующих записей, необходимых для правильного маршалинга. После завершения атрибутов закрывающей квадратной скобкой (]) ключевое слово *interface* начинает описание самого интерфейса.

Мы сделаем *IMotion* двойственным, а *IVisual* — обычным интерфейсом, чтобы показать, как интерфейсы разных типов подключаются к классу *CSpaceship*.

```
[
    object,
    uuid(692D03A4-C689-11CE-B337-88EA36DE9E4E),
    dual,
    helpstring("IMotion interface")
]
```

```
interface IMotion : IDispatch
{
   HRESULT Fly();
   HRESULT GetPosition([out,retval]long* nPosition);
};

[
   object,
   uuid(692D03A5-C689-11CE-B337-88EA36DE9E4E),
   helpstring("IVisual interface")
]
interface IVisual : IUnknown
{
   HRESULT Display();
};
```

После введения описаний интерфейсов заново пропустите IDL-файл через компилятор MIDL для создания новой копии Spaceshipsvr.h с чисто абстрактными базовыми классами *IMotion* и *IVisual*.

Теперь нам надо добавить эти интерфейсы в класс *CSpaceship*. Это делается в два этапа. Первый этап состоит в создании интерфейсной части класса. Давайте начнем с *IMotion*. Добавить интерфейс *IMotion* к *CSpaceship* очень легко — просто используйте шаблон *IDispatchImpl*, предоставляющий необходимую реализацию:

```
class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                      &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
                        &IID_IClassicATLSpaceship,
                        &LIBID_SPACESHIPSVRLib>,
   public IDispatchImpl<IMotion, &IID_IMotion,
                        &LIBID_SPACESHIPSVRLib>
{
⋮
};
```

Второй этап — заполнение карты интерфейсов, чтобы клиент мог запросить интерфейс *IMotion*. Однако наличие в одном COM-классе двух двойственных интерфейсов создает проблему. Запрашивая *IMotion*, клиент должен получить *IMotion*, но когда клиент вызывает *QueryInterface* для *IDispatch*, какую версию этого интерфейса он получит: диспетчерский интерфейс *IClassicATLSpaceship* или *IMotion*?

Множественные двойственные интерфейсы

Все двойственные интерфейсы начинаются с 7 функций интерфейса *IDispatch*. Проблема возникает, когда клиент вызывает *QueryInterface* для *IID_IDispatch*. Как разработчик, вы должны выбрать версию *IDispatch*, передаваемую клиенту.

Интерфейс, возвращаемый *QueryInterface*, задан в карте интерфейсов. В ATL есть особый макрос для обработки двойственных интерфейсов. Сначала рассмотрим имеющуюся карту интерфейсов:

```
BEGIN_COM_MAP(CClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

Когда клиент вызывает *QueryInterface*, ATL проходит по таблице в поисках IID, совпадающего с запрошенным. Приведенная карта содержит два интерфейса: *IClassicATLSpaceship* и *IDispatch*. Если нужно добавить к классу *CClassicATLSpaceship* еще один двойственный интерфейс, то нужен другой макрос.

Для обработки множественных двойственных интерфейсов в ATL существует макрос *COM_INTERFACE_ENTRY2*. Чтобы *QueryInterface* работала правильно, нужно только решить, какую версию получит клиент, запросив *IDispatch*, например:

```
BEGIN_COM_MAP(CClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IMotion)
   COM_INTERFACE_ENTRY2(IDispatch, IClassicATLSpaceship)
END_COM_MAP()
```

В данном случае клиент при запросе *IDispatch* получит указатель на *IClassicATLSpaceship*, первые 7 функций которого являются функциями *IDispatch*.

Добавить недвойственный интерфейс еще проще — добавьте интерфейс в список наследования:

```
class ATL_NO_VTABLE CClassicATLSpaceship :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CClassicATLSpaceship,
                  &CLSID_ClassicATLSpaceship>,
   public IDispatchImpl<IClassicATLSpaceship,
                  &IID_IClassicATLSpaceship,
                  &LIBID_SPACESHIPSVRLib>,
   public IDispatchImpl<IMotion, &IID_IMotion,
                  &LIBID_SPACESHIPSVRLib>,
   public IDispatchImpl(IVisual, &IID_IVisual,
                  &LIBID_SPACESHIPSVRLib>
{
:
};
```

и добавьте запись в карту интерфейсов:

```
BEGIN_COM_MAP(CClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IMotion)
   COM_INTERFACE_ENTRY2(IDispatch, IClassicATLSpaceship)
   COM_INTERFACE_ENTRY(IVisual)
END_COM_MAP()
```

Теперь у вас есть функциональный, работающий, саморегистрирующийся COM-сервер, который поддерживает понятную для COM «игру в компоненты». Но оказывается, в Visual C++ .NET есть другой способ реализации COM-серверов: путем программирования на основе атрибутов

Программирование с применением атрибутов

Вместо обеспечения поддержки COM путем применения шаблонов C++ вы можете прибегнуть к более декларативному подходу — атрибутам. Классическое ATL-программирование предусматривает поддержку *IUnknown* за счет шаблонных классов и макросов карты интерфейсов, а в программировании на основе атрибутов (attributed programming) достаточно объявить COM-класс как таковой прямо в исходном коде.

Создадим тот же сервер Spaceship, но на основе атрибутов. Во-первых, создайте новый ATL-проект, установив флажок Attributed на странице Application Settings мастера ATL Project Wizard. Затем с помощью мастера ATL Simple Object Wizard создайте *класс с атрибутами* (attributed class), назвав его *AttributedATLSpaceship*. Страница Options такая же, что и для классического COM-класса: вы вправе выбрать потоковую модель (Apartment, Free, Both или Neutral), а также поддержку ISupportErrorInfo и/или точек соединения. Однако код, сгенерированный мастером, немного отличается от классического ATL-кода.

```
// IAttributedATLSpaceShip
[
   object,
   uuid("4B8685BD-00F1-4D38-AFC1-3012C786480D"),
   dual, helpstring("IAttributedATLSpaceShip Interface"),
   pointer_default(unique)
]
__interface IAttributedATLSpaceShip : IDispatch
{
};
// CAttributedATLSpaceShip
[
   coclass,
   threading("apartment"),
   vi_progid("AttributedATLSpaceShipSvr.AttributedATL"),
   progid("AttributedATLSpaceShipSvr.AttributedA.1"),
   version(1.0),
   uuid("CE07EBA4-0858-4A81-AD1C-C12710B4A1A2"),
   helpstring("AttributedATLSpaceShip Class")
]
class ATL_NO_VTABLE CAttributedATLSpaceShip :
   public IAttributedATLSpaceShip
{
public:
   CAttributedATLSpaceShip()
   {
   }

   DECLARE_PROTECT_FINAL_CONSTRUCT()
   HRESULT FinalConstruct()
   {
      return S_OK;
   }
   void FinalRelease()
```

```
    {
    }
public:
};
```

Поддержка COM, обеспечиваемая шаблонами на C++ в классическом ATL, привносится в ATL-сервер через *DLL провайдеров* (provider DLL). Заключенные в квадратные скобки атрибуты в начале файла указывают компилятору добавить в класс *CAttributedATLSpaceShip* инфраструктуру COM. Это намного проще, чем отслеживать такие классы (например, *CComObjectRootEx* и *CComCoClass*) и макросы (например, *BEGIN_COM_MAP*).

Дальнейшая разработка COM-класса проста. Допустим, нужно добавить в класс интерфейсы *IMotion* и *IVisible*. В «ATL с атрибутами» довольно просто внести интерфейсы прямо в исходный код ATL, например так:

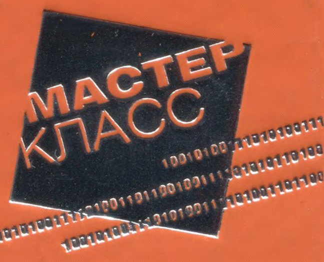
```
[
  object,
  uuid("692D03A4-C689-11CE-B337-88EA36DE9E4E"),
  dual,
  helpstring("IMotion interface")
]
__interface IMotion : IDispatch
{
   HRESULT Fly();
   HRESULT GetPosition([out,retval]long* nPosition);
};
[
  object,
  uuid("692D03A5-C689-11CE-B337-88EA36DE9E4E"),
  helpstring("IVisual interface")
]
__interface IVisual : IUnknown
{
   HRESULT Display();
};
// и так далее…
```

Атрибуты перед ключевым словом *__interface* описывают COM-интерфейсы как таковые, точнее как двойственные интерфейсы. После описания интерфейсов в исходном коде вы можете реализовать интерфейсы класса, просто щелкнув его правой кнопкой в окне Class View, последовательно выбрав в контекстном меню Add и Implement Interface. Вы можете выбирать интерфейсы из зарегистрированных библиотек типов или из интерфейсов, перечисленных в исходном коде (*IMotion* и *IVisual*). Visual Studio .NET создает функции-заглушки — вам достаточно заполнить их текстом.

В результате получается полноценная COM DLL с нужными входными точками: *DllMain*, *DllGetClassObject*, *DllCanUnloadnow*, *DllRegisterServer* и *DllUnregisterServer*.

Программирование Microsoft® на VISUAL C++ .NET

Издание второе





На компакт-диске:

- две оригинальных электронных версии книги: автономная и интегрируемая в справочную систему среды Visual C++ .NET;
- исходные тексты всех приведенных в книге примеров программ.

Самое авторитетное руководство по программированию в среде Microsoft Visual C++ .NET

Эта книга адресована профессионалам, желающим заняться созданием приложений .NET, используя возможности Visual C++ и Microsoft .NET Framework. Вы найдете здесь пошаговые инструкции и подробное обсуждение методов программирования, инструкций и решений, а также рассказ о новинках Microsoft Visual C++ .NET. Вы получите самые подробные сведения о синтаксисе языка, инструментах и API-интерфейсах, а от опытных экспертов — советы и рекомендации по экономии времени и сил при разработке программ.

В книге рассматриваются:

- **Основы:** Windows и Visual C++ .NET, MFC и мастера Visual C++ .NET, классические GDI-функции, диалоговые окна, элементы управления и элементы ActiveX, управление памятью в Win32, обработка сообщений Windows, многопоточные приложения.
- **Архитектура «документ-вид» в MFC:** меню, быстрые клавиши, поля ввода с форматированием и окна свойств, документ и его представление, SDI-, MDI- и MTI-приложения, печать, разделяемые окна и представления данных, контекстная справка, DLL-библиотеки, MFC-программы без классов «документ» и «вид».
- **COM, Automation, ActiveX и OLE:** интерфейс IDispatch, передача данных через буфер обмена и операцией OLE drag-and-drop, ATL, ActiveX-элементы, шаблоны OLE DB.
- **Создание приложений для Интернета:** TCP/IP, Winsock и WinInet, Dynamic HTML, ATL Server.
- **.NET и дальше:** платформа .NET, взаимодействие .NET с управляемым C++, Windows Forms, GDI+ и Web-сервисы на основе C++, Microsoft ADO.NET.

Об авторах

Джордж Шеферд

Кроме создания средств разработки для компании Syncfusion, Джордж Шеферд читает курсы в учебных центрах DevelopMentor. Он выступил соавтором настоящего и нескольких предыдущих изданий этой книги, а также книги «Inside ATL». Джордж написал несколько книг о создании приложений для .NET и о «внутренностях» MFC. Он является пишущим редактором в журнале MSDN Magazine.

Дэвид Дж. Круглински

Один из разработчиков исходной версии Microsoft Visual C++. Погиб при падении дельтаплана в апреле 1997 года.

Издательский дом
Питер
Санкт-Петербург
Б. Сампсониевский пр., 29а
E-mail: sales@piter.com
Internet: www.piter.com
Тел./факс: (812) 103-7383

Издательство
Русская Редакция
Москва
Шелепихинская наб., 32
E-mail: info@rusedit.ru
Internet: www.rusedit.ru
Тел./факс: (095) 256-7145

ISBN 5-469-01178-X



ISBN 5-7502-0270-4



Microsoft®



Программирование на Microsoft® VISUAL C++ .NET

Дж. Шеферд

Microsoft PRESS